Kodoptimering av ett proteinanalysprogram för konservation och kovarians inom ph-domän

André Norrgård

Diplomarbete
Handledare: Jan Westerholm och Mats Aspnäs
Datateknik
Fakulteten för naturvetenskaper och teknik
Åbo Akademi

Abstrakt

I detta diplomarbete har jag analyserat ett datorprogram som är skrivet i programmeringsspråket C. Programmets syfte är att utföra analyser på aminosyresekvenser för att hitta likheter mellan olika typer av aminosyror samt deras positioner i en aminosyresekvens. Programmet beräknar dels entropi och dels ömsesidig information för att välja ut de aminosyror som korrellerar med varandra. Detta delresultat av korrellerande aminosyror används för att konstatera ifall det finns grupper av tre stycken aminosyror som är relaterade. Detta kallas i programmet för triplettanalys. Algoritmernas ursprungliga implementation har optimerats för att minska programmets exekveringstid, RAM-minnesanvändning samt lagringsutrymme vid en körning.

Vanliga sorteringsalgoritmer som bubbelsortering (Bubble Sort) [3, s. 265] har ersatts med quicksort [3, s. 303]. Vikten av att sortera den information som lagrats i räckor och matriser ger en möjlighet att implementera optimerade sökalgoritmer så som binärsökning och exponentiellsökning. Den viktigaste optimeringen som påverkade programmets prestanda var ett antagande som reserverade för mycket minne för att behandla aminosyresekvenserna i minnet. Detta optimerades genom att bevisa hur det går att behandla samma mängd aminosyror med hjälp av att beskriva problemet med mindre minnesanvändning, vilket minskade minneskravet från 3,2 Gb till 110 Mb. Den slutgiltiga prestandaförbättringen gav som resultat att programmet kan köras 112 gånger snabbare, med en minskning av både minnesanvändningen från 3,2 Gb till 37 Mb och diskutrymme från 277 Mb till 21 Mb.

Sökord: Proteinanalys, Korrelation, Aminosyresekvens, Optimering, Triplett.

Innehållsförteckning på tabeller, förkortningar och viktiga variabler

Covana:
Namn på mjukvaruprogram skrivet i C-programmeringsspråket. Covana.cpp är namnet på den fil som innehåller programmet och som har blivit optimerat.

Körning av Covana.cpp samt parametrar:
-$ Covana filename.fasta gap_percentage pval1 pval2

filename.fasta:
Namn på fil i FASTA-format, innehållande proteinsekvenser. ex. 7tm5.fasta eller Herpes_MCP.fasta

Gap_percentage:
Procentuellt tillåtet antal gap-tecken i en sekvens som tas med i uträkningen, angiven i decimaltalsformat. Exempelvis: 0.21 betyder 21 %.

Pval1:
Sannolikhetsgräns för förekomsten av aminosyror som ska tas med i uträkning. Exempelvis: 0.04

Pval2:
Sannolikhetsgräns för förekomsten av aminosyror som ska tas med i uträkning. Exempelvis: 0.2

Exempel på kommandorad för en körning av Covana programmet:
-$ covana Herpes_MCP 0.21 0.04 0.2

KK:
Antal aminosyror som analyseras som triplettkandidater i filen "Herpes_MCP.fasta". 28441

seqnum:
Antal aminosyresekvenser i filen "Herpes_MCP.fasta": 90

resnum:
Antal tecken per sekvens i filen "Herpes_MCP.fasta": 1520

Topnum:
Antal par med ett ömsesidigt informationsvärde över gränsen för att tas med till triplettanalys från filen "Herpes_MCP.fasta": 14364

Triplets:
Antalet trerelaterade aminosyrepar i filen "Herpes_MCP.fasta": 8596

Formler:

Teoretiska maximiantal par av aminosyror i en sekvens:
N=length*(length-1)/2, med length = 1520 ger N = 1 154 440.
(Length, length0: sekvenslängd eller antal tecken per sekvens. Samma som **resnum**.)

Kkestimate: variabel som räknar det antal par av aminosyror som kvalificerar sig för triplettanalys. Max storlek är resnum * resnum.

MaxResN: Värdet 5000, maximiantal tecken i en FASTA-fil för analys.
MaxSeqN: Värdet 5000, maximiantal sekvenser av aminosyror.

Innehållsförteckning

1. Inledning

I avhandlingen behandlas prestandan i ett proteinanalysprogram som utför korrelation och kovariansanalys av aminosyror. Programmet har skapats av Bairong Shen och Mauno Vihinen som år 2005 arbetade vid Tammerfors universitet [2] [8] [13] [14]. Orsaken till att programmet optimerades var att det krävde mycket resurser för att kunna slutföra en analys. I vissa körningar reserverades en stor del av det tillgängliga RAM-minnet för att utföra en analys och körtiden varierade från några minuter till att vissa gånger resultera i en programkrasch. En användare av programmet valde startparametrar för analysen och vissa inparametrar resulterade i att programmet inte kunde slutföra analysen utan kraschade. Detta var en orsak varför det var önskvärt att optimera programmets prestanda så det går att utföra analyser snabbare samt att slutföra de analyser som inte gick att köra.

Forskning inom biologi med hjälp av datorberäkningar hör till bioinformatik. Inom bioinformatiken analyseras molekylärbiologiska data som till exempel dna eller proteiner [1]. Den minsta byggstenen inom ett protein är aminosyran [1] [6]. Det finns 20 stycken olika aminosyror som kan delas in i mindre grupper utgående från deras egenskaper. En grupp kan innehålla aminosyror som är elektriskt negativt laddade, positivt laddade eller har hydrofobiska egenskaper, (se kapitel 2.1).

Proteiner är uppbyggda av aminosyresekvenser av varierande längd. För att analysera ett protein inom bioinformatiken används filer i FASTA-format [1]. En fil i FASTA-format innehåller en översättning av proteinets aminosyror i teckenformat för att sekvensen ska kunna behandlas datatekniskt [1].

1.1 Avhandlingens ändamål

Diplomarbetet har gjorts i samarbete med Åbo Akademi och Tammerfors universitet. Bioinformatikavdelningen vid Tammerfors Universitet har utvecklat ett program ”Covana” som använder sig av flera sekvenser av aminosyror för att analysera korrelation och kovarians. Programmet har utvecklats inom ett projekt som kallas för ProCon. Programmet är avsett att ge en användare möjligheten att utföra analyser via internet [2].

Korrelationsanalysen mellan två proteiner utförs genom att indexera och gruppera var och en aminosyra inom sekvensen. Sannolikheten att en aminosyra förekommer på en viss position registreras. Om sannolikheten överstiger ett gränsvärde är aminosyran en kandidat för triplettanalys. En triplett betyder att det inom sekvensen hittas tre aminosyror som korrelerar med varandra

Det ursprungliga programmet "Covana" krävde för mycket datorkapacitet, såsom processorkraft och minne vid en normal körning. Detta är en av orsakerna till att programmet togs med i FinHPC-projektet för optimering. FinHPC finansierades delvis av Tekes och idag heter organisationen Business Finland. Programmet ”Covana” krävde vid vissa körningar så mycket minne och processorkraft att en metod som kunde vara ett alternativ att optimera körningen var att skriva om programmet med hjälp av parallellprogrammering. Parallellprogrammering skulle tillåta att flera processorer används för att hjälpa till med tunga uträkningar, detta skulle kunna realiseras till exempel med hjälp av Message Passing Interface [15].

1.2 Strukturering och ändamål

Diplomarbetets syfte är att optimera programmet Covanas prestanda; med optimering avses att förbättra exekveringstid, RAM-minnesanvändning samt filhantering. Det ursprungliga programmet har omformats stegvis för att kunna mäta prestandan efter varje ändring. För att kunna jämföra mätresultat har programmet versionerats och körts under tidtagning. Programmets minnesanvändning såsom RAM samt hårdskiveutrymmesmätningar har också registrerats. Detta hjälper till att förstå hur de optimeringar som har gjorts, i respektive version av programmet, påverkar dess effektivitet.

Arbetets upplägg består av att först diskutera de algoritmer och datastrukturer som befann sig i det ursprungliga programmet samt vilka algoritmer och datastrukturer som testats under utvecklingens lopp och sedan blivit valda i den slutgiltiga versionen av programmet. Först beskrivs det ursprungliga programmets egenskaper och problemområden. Sedan förklaras varje optimering som utfördes per version. Versionsförklaringarna innehåller en beskrivning av vilken optimering som har

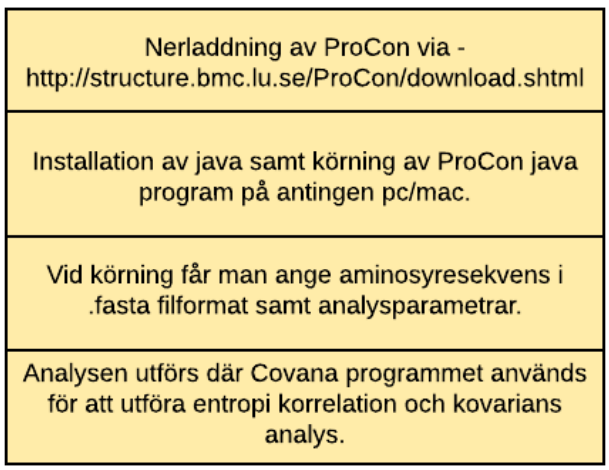
gjorts samt den versionens exekveringstid och minnesanvändning. Till sist jämförs versionernas mätresultat med hjälp av grafer för att ge en överblick i de optimeringar som bidragit med olika förbättringar.

1.3 FinHPC - Business Finland

FinHPC är ett av flera projekt som understötts av Business Finland. Tekes är det gamla namnet, som användes 2005-2006 då detta optimeringsarbete utfördes, på den organisation som idag heter Business Finland [17]. FinHPC projektets mål är att analysera och optimera vetenskapliga projekt som används på CSC datorer. FinHPC är ett samarbetsprojekt med Åbo Akademi och CSC. De program som kräver stor datorkapacitet samt exekveringstid för att exekveras är goda kandidater till FinHPC projekt. ProCon projektet har utvecklats i Tammerfors universitet och FinHPC har valt att inkludera detta. Tekes har sponsrat mitt slutarbete där jag har optimerat kärnprogrammet i ProCon projektet med hjälp av mina handledare Jan Westerholm och Mats Aspnäs.

1.4 ProCon

ProCon har utvecklats av Bairong Shen och Mauno Vihinen vid Tammerfors universitet. Projektets syfte är att ge möjligheten till en användare att jämföra aminosyresekvenser för att få reda på korrelation, ömsesidig information och tripletter [8].



Figur 1 Beskrivning av hur det optimerade covana programmet används idag.

Exempelvis kan en användare navigera till en sida på nätet där användaren har möjlighet att ange den aminosyresekvens som ska analyseras i form av FASTA-filformatet, (se kapitel 2.5). Användaren tillåts definiera nödvändiga parametrar som används i analysen. Den del av programmet eller Covana.cpp, som jag har optimerat kan då köras när användaren väljer att starta analysen. Proteinanalys utförs i C-programmet som heter Covana.cpp. Den websida som beskriver ProCon projektet i referens [2] baserar sig på de optimeringar jag gjorde i detta arbete. Däremot beskrivs det på ProCon websidan [2] att programmet består av Java. År 2006 då jag utförde detta optimeringsarbete utförde jag optimeringarna i C samt C++ kod, optimeringarna kan ännu idag vara gällande för det program som finns på ProCon websidan [2]. Huvudsakligen har de optimeringar jag gjort koncentrerats till filen Covana.cpp, (se kapitel 3 och 4). För att bättre förstå, vilka analyser som används samt att få en överblick över hur programmet fungerar, kommer jag kort att beskriva aminosyror samt vilka analyser som används i Covana programmet.

2. Material

Covana är huvudberäkningsprogrammet där de mest datorkapacitetskrävande uträkningarna utförs. Analyser av ömsesidig information, sannolikheten för att en viss aminosyra förekommer på en viss position räknas ut samt beräkning av tripletter.

Det ursprungliga programmet är skrivet i C/C++ och för att köra programmet ska användaren i kommandoterminalen ange 4 argument. Argumenten består i ordningsföljd av:

- Datafil i FASTA-format innehållande aminosyresekvenser.
- Gap-procent.
- pvalue1.
- pvalue2.

Gap-procenten anger vilket antal tomma platser som analysen ska tolerera. Pvalue1 och pvalue2 anger sannolikhetsgränser för vilka par av aminosyror som skall

beaktas i triplettsökningen. Efter en körning av Covana.cpp programmet skapas en del filer för att sammanställa resultatet, detta resultat kan användas av andra program för visualisering.

2.1 Grupper av aminosyror

I min kandidatavhandling [1] beskriver jag med hjälp av referenser från Gentekniknämnden [6] samt Collins lexikon [7] att aminosyror är byggstenar som bygger upp dna på det viset att följden på aminosyrorna beskriver olika proteiner. I den studie som Bairong Shen och Mauno Vihinen har publicerat [8] har de valt att använda 6 stycken grupper av aminosyror som har likadana egenskaper [8]. En grupp av aminosyror är till exempel vattenavstötande, för en beskrivning varför 6 stycken grupper av aminosyror har valts kan förstår genom att läsa "*Conservation and covariance in PH domain sequences: physicochemical profile and information theoretical analysis of XLA-causing mutations in the Btk PH domain*" [8].

I det ursprungliga programmet Covana användes följande tecken för att representera aminosyrorna:

```
char residue[22]="ACDEFGHIKLMNPQRSTVWY-";
```

Figur 2 Kodutdrag av definition av aminosyror

I den ursprungliga versionen representerades de 6 olika grupperna av aminosyror med hjälp av följande programkod, se Figur 3.

2.2 Entropi

I Covana räknas entropi ut i funktionen ProbabilityCalc(), formeln beskrivs nogrannare i den artikel som publicerats av Bairong Shen och Mauno Vihinen [8], se figur 4.

```
for(i=0; i<length0; i++){
    for(j=0; j<7; j++){
      prob6aa[i][j]=0;
      //j=0 C F I L M V W Y
      //j=1 A T
      //j=2 D E
      //j=3 G P
      //j=4 N Q S
      //j=5 H R K
      //j=6 -
    }
  }
```

Figur 3 Kodutdrag ur den ursprungliga versionen beskrivande grupper av aminosyror.

$$E_{entropy} = -\sum_{ai} P_{ai} * \log(P_{ai})$$

Figur 4: Entropiekvation [8, s. 268],

$$Information = \log N + \sum_{ai} P_{ai} \log(P_{ai})$$

Figur 5 Entropiinformationsekvation [8, s. 271]

I formlerna ovan står P_{ai} för sannolikheten att en aminosyra existerar på position i, i en sekvens [8]. Resultatet av entropiuträkningen sparas i filer. Entropianalysen visualiseras grafiskt i ett annat program inom ProCon projektet.

2.3 Ömsesidig information

Ömsesidig information, (eller Mutual Information på engelska) beräknas i funktionen MutualInformation() i Covana programmet. Formeln beskrivs i en publikation av Bairong Shen och Mauno Vihinen [8]. I huvudsak används följande formel för att definiera uträkningen av ömsesidig information:

$$M_{mutual\ \inf ormation} = \sum_{ai,aj} (P_{ai,aj})^2 * \log\left(\frac{P_{ai,aj}}{P_{ai} P_{aj}}\right)$$

Figur 6 MI formel från en publikation av Bairong Shen och Mauno Vihinen [8, s. 268]

I figur 6 ger $P_{ai,aj}$ ett värde för hur sannolikt aminosyra a förekommer på position i och j [8]. P_{ai} anger sannolikheten för att aminosyra a förekommer på position i [8].

Ett av de huvudsakliga effektivitetsproblemen hittades i den algoritm som utförde uträkningen av ömsesidig information. Ömsesidig information innebär hur mycket ett par av aminosyror korrelerar inuti ett protein. Formeln nedan används för att räkna ut ett teoretiskt maximiantal par som kommer att jämföras.

```
N=length0*(length0-1)/2
```

Figur 7 Kodutdrag av maximalt antal par av aminosyror i en sekvens. [10, Line 1174]

N: Antal möjliga ömsesidig informationspar.
Length0: Antal tecken (aminosyror) i FASTA-filformat, samma som variabeln resnum.

Den fil i FASTA-format som användes i testningen bestod av 1520 tecken långa proteiner. Det vill säga att sekvensen var av längden 1520. Genom att använda formeln i figur 7 ovan kommer det teoretiska maximiantalet par att vara 1 154 440.

Då ömsesidiga informationsvärdet räknas ut för varje par av aminosyror i sekvensfilen så kvalificerar sig inte alla N aminosyrepar. I den körning som använde sig av Herpes_MCP.fasta var en sekvens av längden 1520 och med följande inparametrar:

- Gap_percentage = 0.21
- pvalue1 = 0.04
- pvalue2 = 0.2

kvalificerade sig 28441 par av aminosyror för triplettsökning. Detta värde sparas i variabeln KK och användes för att reservera minne för datastrukturer relaterade till triplettsökning.

Som beskrivet ovan innebär KK variabeln det antal aminosyror som kvalificerat sig till triplettuträkning. Variabeln KK var också relaterad till det huvudsakliga optimerings problemet i funktionen. Det ursprungliga Covana programmet sparar ner information om varje ömsesidigt informationspar som identifierats. Informationen lagrades i en ny fil per identifierat ömsesidigt informationspar. Filens format definierades som följande: ”filnamn.pij._i#_j#”. Där p står för sannolikhet (Probability) och i samt j anger positionerna för varje aminosyra i paret.

Texten #i byttes ut mot den första aminosyrans numeriska position inom sekvensen och #j byttes ut mot den andra aminosyrans position. En körning av programmet resulterade i att tusentals filer med mycket litet innehåll i varje fil. Detta var ett av de huvudsakliga problemen som korrigerades i en optimering av Covana, (se kapitel 4.2).

Skapandet av filerna gjordes inuti en slinga där varje iteration inkluderade andra uträkningar. En filoperation är långsammare än en minnesoperation. Åtkomsttiden för filoperationer till en Seagate hårdskiva av modellen ST8000DM002 ligger på 4,16 millisekunder [18]. Jämförelsevis har ett Kingston DDR4 RAM minne med 3200 Mhz klockfrekvens en CL16 vilket ger en åtkomsttid på 5 nanosekunder [19]. Minnesoperationer i detta fall kan vara snabbare på en faktor upp till 1 000 000 gånger. Detta är också en bidragande faktor till varför det tog länge att exekvera programmet.

2.4 Triplett

En triplett i Covana programmet innebär att tre stycken aminosyror som korrelerar med varandra identifieras. Alla aminosyror i en triplett ska vara relaterade med varandra för att konstateras vara en triplett.

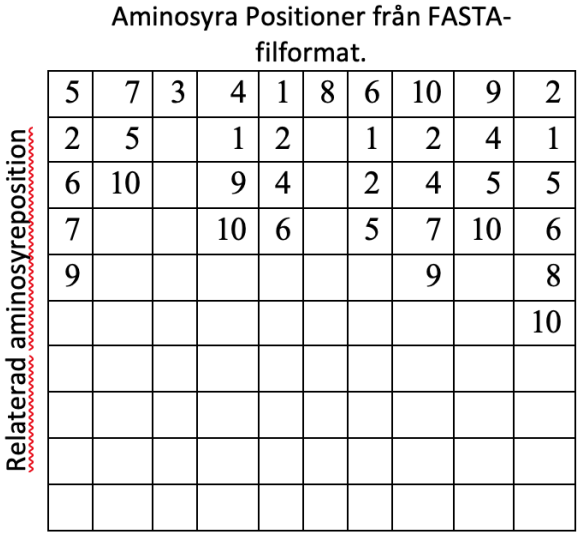
Här följer ett exempel med aminosyrorna A, B och C där var och en aminosyra har sin unika position inom en aminosyresekvens. För att en triplett ska identifieras behövs följande regler gälla:

-A måste korrelera med B
-B måste korrelera med C
-C måste korrelera med A

I exemplet kan inte följande antas, att ifall A korrelerar med B och B korrelerar med C så korrelerar också A med C. Antagandet A - C måste identifieras på samma sätt som de andra relationerna inom tripletten.

Korrelationerna mellan aminosyrorna räknas ut i funktionen MutualInformation(). Som resultat av att funktionen MutualInformation()har körts till slut bör en matris

med alla aminosyror som korrelerar med varandra finnas. Informationen sparas ner till en datastruktur som kan liknas vid en matris, se figur 8.



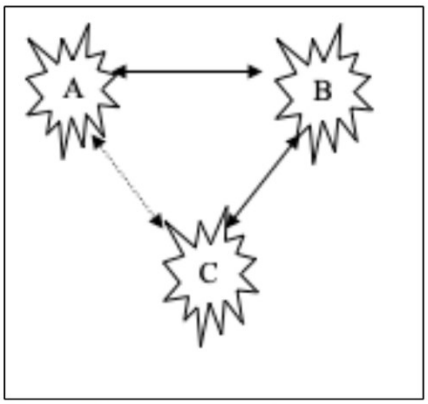
5	7	3	4	1	8	6	10	9	2
2	5		1	2		1	2	4	1
6	10		9	4		2	4	5	5
7			10	6		5	7	10	6
9							9		8
									10

Figur 8. MuiSite matrisen.

Datastrukturen i det ursprungliga Covana programmet består av en dubbelräcka med datatypen integer. I dubbelräckan eller matrisen sparas alla relationer av aminosyrepositioner som identifierats från ömsesidiga informationsuträkningen. Se figur 8, på första raden beskrivs positionerna för den första aminosyran. I kolumnerna finns de andra aminosyroronas positioner. En aminosyra kan vara relaterad med flera andra aminosyror och därför fylls kolumnerna med positionerna av den andra aminosyran i ett relaterat par aminosyror.

För att illustrera ett exempel av en triplett. Låt oss säga att vi har tre relaterade par aminosyror:

A(1,2)
B(2, 3)
C(3, 1)



Figur 9 Illustration av en triplett.

2.5 Filformatet FASTA

Filformatet som innehåller information om proteiner eller aminosyresekvenser heter FASTA.

```
>ABVSD_IWODN/6-213
AAAABBBBBCCCCDDDDWWWWEEEAAAWWWWAA-AAAABBBBBCCCCDDD-AAAABBBBB
AAAABBBBBCCCCDDDDWWWWEEEAAAWWWWAA --AAAWWWWAAA---VVVVA----AA
AAAABBBBBCCCCDDDDWWWWEEEAAAWWWWAA-AAAABBBBBCCCCDDD-AAAABBBBB
AAAABBBBBCCCCDDDDWWWWEEEAAAWWWWAA --AAAWWWWAAA---VVVVA----AA
AAAABBBBBCCCCDDDDWWWWEEEAAAWWWWAA-AAAABBBBBCCCCDDD-AAAABBBBB
AAAABBBBBCCCCDDDDWWWWEEEAAAWWWWAA --AAAWWWWAAA---VVVVA----AA
```

Figur 10. Exempel på ett FASTA-filformat, teckensekvensen påhittad.

Det finns 20 aminosyror som i FASTA-filformatet representeras av olika bokstäver, (se kapitel 2.1). I figur 10 finns ännu ett tecken "-", tecknet används för att representera en tom plats i proteinsträngen. När en analys av en proteinsträng görs med Covana programmet anges en gräns för hur stor del av gap tecken som får tas med i analysen. Fasta-filformatet är ett allmänt filformat som används för att representera aminosyresekvenser. Detta filformat beskrivs i min kandidatavhandling i kapitel 3.2 [1].

3. Programanalys

All den kod som den ursprungliga versionen av programmet bestod av var implementerat i en enda stor huvudfunktion kallad main(). På grund av detta var

koden aningen svårläst men det fanns kommentarer i koden som hjälpte till att förstå innebörden. De datastrukturer som användes i programmet var datatyper som double, int, char samt räckor och dubbelräckor av tidigarenämnda datatyper. Objekt hade inte använts och ingen uppdelning i funktioner för att ordna ihop helheter i programmet.

Slingor används för att iterera igenom räckorna samt switchar och villkorssatser (på engelska if statement) för att göra beslut. Med tanke på minnesanvändningen kan datastrukturer bytas ut mot andra effektivare datastrukturer. Med tanke på programmets prestanda behöver också minnesallokeringen granskas. Den utdata som som programmet producerar behöver också granskas för att se ifall där finns redundant eller överflödiga data som kan skalas bort. Ju mindre information programmet behöver behandla desto snabbare kommer programmet att utföra analysen.

3.1 Datastrukturer i Covana

I datorprogram definieras variabler av olika typer av datastrukturer. Variabeln används för att hänvisa till en minnesadress och datastrukturen definierar vilka typer av data som kan representeras samt hur mycket minne variabeln behöver reservera. När en datastruktur definieras och programmet exekveras, leder detta till att programmet reserverar RAM-minne för att sedan kunna lagra ett värde för att användas i andra delar av programmet. Värdet representeras av bitar i multipler av 8, där vi har 8 bitar som motsvarar 1 byte. Den minsta datatypen är char eller character, den använder en byte minne och double hör till den största datatypen som använder sig av 64 bitar eller 8 byte. I boken C++ primer [5, s. 36, tabell 2.1] beskrivs alla primitiva datatyper i c++

3.1.1 Lista över datastrukturer

Lista över datastrukturer i det ursprungliga programmet kan ses i kapitel 3.1.1.1 jämfört med datastrukturer i det optimerade programmet som kan ses i kapitel 3.1.1.2.

3.1.1.1 Lista över viktiga datastrukturer i det ursprungliga programmet

Räckor:

- Muinf12 - double datatyp, räcka av storlek [N].

Dubbelräckor eller Matriser:

- Rankaa - char datatyp, dubbelräcka av storlek [length0][21]
- Rankpc - char datatyp, dubbelräcka av storlek [length0][15]
- Seq - char datatyp, dubbelräcka av storlek [MaxSeqN][MaxResN]
- Probprob - double datatyp, dubbelräcka av storlek [20][20]
- Probaa - double datatyp, dubbelräcka av storlek [length0][21]
- Probpc - double datatyp, dubbelräcka av storlek [length0][15]
- Prob6aa - double datatyp, dubbelräcka av storlek [length0][7]
- mutuinf - double datatyp, dubbelräcka av storlek [length0][length0]
- freqaa - double datatyp, dubbelräcka av storlek [length0][21]

3.1.1.2 Lista över viktiga datastrukturer i det optimerade programmet

- MuiSite - fyra stycken unsigned int (short).
  - Storlek sizeof([struct MuiSite)*resnum).
- MutObj - struct med en double, två integers.
  - Storlek sizeof(struct Mutobj)*kkestimate.
- MutuInf - En double typ matris.
  - Storlek resnum*resnum*sizeof(double) storlek.
- Triplet - struct med 3 shorts.
  - Storlek tpsize = **int**(((resnum*resnum*resnum) - 3*(resnum*resnum) + 2*resnum)/6 * 0.002);
- Rankprobpc - struct en double en char.
  - Matris av storlek: resnum*resnum*15.
- Rankprobaa - struct en double en char.
  - Matris av storlek: resnum*resnum*21.

3.1.2 Primitiva datatyper

Några av de primitiva datatyper som används i Covana hör till följande:

- char - Character (8 bit / 1 Byte)
- short - Short integer (16 bit / 2 Byte)
- unsigned int - Integer (32 bit / 4 Byte)
- int - Integer (32 bit / 4 Byte)
- double - flyttal. (64 bit / 8 Byte)

Storleken för vissa datatyper beror på den processor som används för att köra programmet [5, s. 35]. Den processor som användes år 2005/2006 för att optimera covana programmet reserverade 4 byte för en integer.

3.1.3 Icke-primitiva datatyper

Icke-primitiva datatyper består av egendefinierade datastrukturer som kombinerar ett antal primitiva datatyper till en ny struktur. I C++ kan egna strukturer definieras med hjälp av anropet struct, strukturen kan användas genom att allokera minne, se figur 11 nedan. Vid minnesallokering för en variabel av datatypen rankprobpc strukturen kommer variabeln att reservera 8 byte för double samt 1 byte för char datatypen. Resultatet blir att totalt 9 byte reserveras i minnet.

```
struct rankprobpc {
  double probpc;
  char rankpc;
};
```

Figur 11 Kodutdrag av egendefinierad datastruktur.

Det ursprungliga programmet innehöll inte någon egendefinierad datastruktur utan använde sig av räckor och matriser för den datarepresentation som behövdes.

3.1.4 Minnesallokering

Datastrukturer som används i ett datorprogram behöver allokera en tillräckligt stor minnesplats för att spara ner den information som ska representeras i programmet. Beroende på den processorarkitektur som programmet körs på kommer olika primitiva datatyper att ta olika mycket plats [5, s. 35]. För att räkna ut hur mycket minne en variabel använder sig av kan kommandot *sizeof(variable)* användas. Resultatet anges i bytes [5, s. 167]. På detta sätt kan storleken av både primitiva- samt icke-primitiva datastrukturer räknas ut.

Enligt optimeringsguiden för AMD Athlon och Opteron processorer [11, s. 91] rekommenderas att objekt allokeras i minnet med ett visst intervall. Intervallet beror

på de datatyper som objektet innehåller. En naturlig uppställning, "*Natural Alignment*" är enligt optimeringsguiden [11, s. 91] definierad så att ett objekt ska finnas på en minnesadressplats som är en multipel av dess storlek.

I en online artikel av Sumedh N. från Intel [20] förklaras att padding kan förekomma vid minnesallokering för vissa strukturer. Om strukturernas datatyper inte är en multipel av deras storlek så kan mera minne allokeras för att uppnå en naturlig uppställning.

Minnesallokeringen i C++ kan utföras med hjälp av uttryck såsom malloc, calloc, realloc eller new [5, 145]. Det ursprungliga programmet använde sig av C++ instruktionen new istället för C-bibliotekets malloc.

```
try {
    seq = new char*[MaxSeqN];
    for (k=0; k<MaxSeqN; k++)
        seq[k]= new char[MaxResN];
  } catch (...) {
    cout<<"Exception raised: for seq"<<"\n";
    return 1;
  }
```

Figur 12 Exempel på minnesallokering i det ursprungliga programmet Covana.

Figur 12 visar ett exempel på hur minnesallokeringen utfördes. Här allokeras minnet i flera steg för en matris. Variabeln seq representerar en räcka med datatypen char med längden MaxSeqN, en slinga itererar MaxSeqN antal gånger för att allokera en ny räcka med längden MaxResN i varje iteration. Detta sätt att allokera minne i små delar åt gången kan vara ett problem effektivitetsmässigt. Ifall minne allokeras i små delar kan minnet fragmenteras till flera olika platser i RAM-minnet. Med tanke på effektivitet eftersträvas att allt minne allokeras på en och samma gång så att minnesreserveringen är kontinuerlig.

Figur 13 är ett exempel på hur en minnesreservering kan göras med hjälp av kommandot malloc. Malloc tar som inparameter det antal bytes som ska reserveras i minnet. Då minne reserveras för datastrukturer är det fördelaktigt att reservera allt minne som behövs på en och samma gång. I det ursprungliga programmet används operatorn ”new” för att initialisera räckor och variabler.

Se Figur 13 för ett exempel av den optimerade Covana koden:

```
rankprobpc **rppc = (struct rankprobpc **)malloc(resnum*sizeof(struct rankprobpc*));
rankprobpc *RPPC = (struct rankprobpc *)malloc(resnum*15*sizeof(struct rankprobpc));
for (i=0; i<resnum; i++) {
  rppc[i] = RPPC + i*15;
  for (j=0; j<15; j++) {
    rppc[i][j].probpc = 0.0;
    rppc[i][j].rankpc = aa_pc[j];
  }
}
```

Figur 13 Minnesallokering av en dubbelräcka eller matris av datastrukturen rankprobpc. Se Figur 11 för deklarationen av datastrukturen rankprobpc

3.2 Informationsbehandling

Räckan muinf12 var den enda räcka som sorterades, med hjälp av bubbelsorteringsalgoritmen, se kapitel 3.2.1.1, förutom räckan var informationen som sparades i räckorna inte sorterad. Att sortera information i räckorna är i allmänhet bra eftersom det ger tillgång till snabbare och effektivare sökalgoritmer som binärsökning. Triplettalgoritmen använder sig av den mutualinformation genom att söka igenom och försöka hitta relationer mellan aminosyror. Detta innebär att det kan vara av intresse att se till att effektiva sökalgoritmer kan användas för triplettsökningen.

3.2.0 Estimering av beräkningstid

Att estimera beräkningstiden för en algoritm kan uttryckas med hjälp av Stora-O eller ”Big-O” metoden. Detta innebär att en estimering av den tid som en algoritm kräver då den körs på en stor mängd data kan fås. Denna metod beskrivs i boken *Introduction to the theory of Computation* [21, s. 226]. Genom att använda Stora-O metoden beskrivs endast den del av koden som kräver störst mängd tid att exekvera. Låt oss anta att vi har en programkod innehållandes två nästade slingor som itererar över samma mängd N element. Den första nästade slingan är av andra graden, och den andra nästade slingan är av tredje graden, båda slingorna itererar över samma mängd N element. Genom att använda Stora-O metoden estimeras körningstiden till $O(N^3)$. Med hjälp av Stora-O metoden framstår endast de mest betydande delarna av ett program, algoritm eller matematisk formel [3, s. 44].

3.2.1 Sorteringsalgoritmer

En stor del av de program som körs på en dator använder en stor del av processorkraften till att söka efter information, detta är fallet i Covana. Detta är också en orsak varför det finns ett antal olika algoritmer tillgängliga för att söka efter information. Ett preliminärt krav för att kunna använda effektiva sökalgoritmer på en mängd information är att informationen måste vara sorterad. Om informationen inte består av en sorterad mängd så innebär det att varje element i mängden ska itereras igenom för att hitta en möjlig likhet.

3.2.1.1 Bubbelsortering

Det ursprungliga programmet använde sig av sorteringsalgoritmen bubbelsortering (Bubble Sort på engelska). Bubbelsortering är enkel på det sätt att den jämför varje element i en räcka för att sedan byta plats på elementen enligt sorteringskriterier. I boken *Algorithms in C* [3, s. 265] beskrivs algoritmen att ha två stycken nästade slingor vilket resulterar i en exekveringstid av $O(N^2)$. Följande kodutdrag i figur 14 visar det ursprungliga programmets implementation:

```
for (i=0; i<length0; i++){
  for (j=0; j<21; j++){
    for (k=j+1; k<21; k++){
        if (probaa[i][j] < probaa[i][k]) {
            temp1=probaa[i][j];
            probaa[i][k]=temp1;
        }
    }
    //cout<<rankaa[i][j];
  }
```

Figur 14 Kodutdrag från den ursprungliga filen covana.cpp exempel av bubbelsortering.

3.2.1.2 Quicksort

Quicksortalgoritmen finns implementerad i ett bifogat C++ bibliotek till exempel cstdlib.h [10] i C-distributionen, denna distribution användes för att optimera Covana. Bubbelsortering, den tidigare sorteringsalgoritmen som i en körning i värsta fall estimeras ta upp till $O(N^2)$, förbättrades genom att byta till Quicksortalgoritmen. Quicksortalgoritmen kan i värsta fall använda $O(N^2)$ operationer men i medeltal använder den sig av O(N LogN) operationer [3, s. 303].

3.2.2 Sökalgoritmer

Sökalgoritmers syfte är att hitta information i en blandad samling med data. Datorn används i allmänhet som informationsbehandlingsredskap eftersom den kan hitta information i en stor mängd lagrade data mycket snabbare än vad människan förmår. Många sökalgoritmer har utvecklats och jag kommer att gå igenom de sökalgoritmer som har använts i det ursprungliga programmet Covana samt de algoritmer som introducerats då programmet optimerats.

3.2.2.1 Sekventiell sökning

Den första versionen av programmet använde sig av sekventiell sökning (sequential search på engelska). Sekventiell sökning innebär att varje element i en räcka gås igenom från början till slut för att hitta en matchning. Enligt boken *Algorithms in C* [3, s. 494] kräver sekventiell sökning i värsta fall att varje element N gås igenom. Låt oss anta att antalet N beskriver antalet element i en räcka, det innebär att i värsta fall kan hela räckan behöva gås igenom för att hitta nyckelelementet. I medeltal är söktiden N/2 [3, s. 493].

Sekventiell sökning är fördelaktigt av den orsaken att algoritmen är enkel att implementera och den fungerar på en osorterad mängd data. Då varje element i en räcka gås igenom för att hitta ett värde, behöver inte en sorterad mängd användas. Det finns däremot andra effektivare sökalgoritmer som kan användas eftersom deras söktider är snabbare än söktiden för sekventiell sökning.

Sekventiell sökning är den sökalgoritm som används i den ursprungliga versionen av programmet Covana. Algoritmen för att leta efter tripletter använde sig av sekventiell sökning inuti en fjärde gradens nästad slinga.

3.2.2.2 Binärsökning

Binärsökning (Binary Search på engelska) är en algoritm som följer söndra och härska-strategin (Divide and conquer på engelska). Algoritmen opererar på en sorterad mängd och har därmed fördelen att göra ett val åt vilket håll den ska göra nästa sökning. Algoritmen delar upp mängden till hälften för varje iteration och ser ifall värdet har hittats eller om den ska fortsätta söka i den vänstra eller högra uppdelningen. På så sätt fortsätter algoritmen tills den antingen hittar det element som söks eller att den inte hittar värdet. Enligt boken *Algorithms in C* [1, 501] i värsta fall har binärsökningsalgoritmen en söktid på log(N), dvs. logaritmiska värdet av alla element.

3.2.2.3 Exponentiellsökning

Exponentiellsökning är en sökalgoritm som använder en sorterad mängd för att hitta en nyckel. Istället för att dela upp mängden på hälften så som binärsökningsalgoritmen gör, så utförs indexeringen exponentiellt. I exponentiell sökning ökas söknyckelns indexvärde exponentiellt för att se om värdet som söks finns på den vänstra eller den högra sidan. Om värdet inte är på den vänstra sidan betyder det att algoritmen fortsätter söka efter värdet på den högra sidan. Indexvärden ökas exponentiellt i följande ordning: 0, 1, 2, 4, 8, 16, 32, 64, 128, 256 osv.

Om exponentiellsökning används för att söka igenom en sorterad räcka på N-element så kommer antalet sökningar i värsta fall kräva N-jämförelser.

Den första implementationen av exponentiellsökning visade att algoritmen var långsammare än binärsökning. Tanken bakom att använda exponentiellsökning var att det snabbare skulle kunna gå att avgränsa ett mindre område för fortsatt sökning. Den slutgiltiga versionen använder sig av en kombination av exponentiell sökning samt binärsökning. Först används exponentiellsökning för att avgränsa ett område i räckan för att sedan finkammas med hjälp av binärsökning. Implementation beskrivs i en kurs i ämnet ”Computational Geometry” [22] som baseras på en bok med samma namn [23]. Den estimerade tiden som algoritmen kräver är O(log N) [22][23].

3.3 C/C++-optimeringar

De optimeringar som beskrivs i kapitel 3.3 hör till processor specifika C/C++ optimeringar som har använts i Covana koden. Optimeringarna beskrivs i detalj i guiden *Software Optimization Guide for AMD Athlon™ 64 and AMD Opteron™* [11].

3.3.1 Uppveckling av slinga

Exempel på uppveckling av slinga (Loop unrolling på engelska) finns i kapitel 4.16.1. Optimeringsmetoden beskrivs i *Software Optimization Guide for AMD Athlon* [11, s. 13]. Genom att ersätta en slinga som har ett litet antal iterationer med explicit kod kan göra program snabbare. Detta beror på att det kan gå åt mera processortid att utföra en slinga än att explicit utföra de operationer som slingan representerade. Se referens [11, s. 13] för mera information.

3.3.2 Ordning av switch fall i mest frekventa fallen först

Exempel på att ordna de fall som har störst sannolikhet att förekomma i en sekvens i storleksordning gör att det begränsar antalet kontroller som görs [11, s. 28]. Exempel på detta finns i kapitel 4.12.

4. Utförda optimeringar per version

Optimeringen skedde i små delsteg där varje optimering efter implementation kördes och tid samt minnesanvändning sparades. Förbättringarna kommer att beskrivas i detalj nedan med en referens till den version där optimeringen påverkade testresultatet. För att få en överblick över hur länge programmetskörtid, minnesanvändning och diskutrymmesanvändning ser ut per version var god se kapitel 5.1 - Mätresultat.

En del versioner har blivit noterade men kan sakna tidtagning. Orsaken till detta beror på att det ursprungliga programmets struktur har ändrats i hopp om att få koden mera strukturerad. En annan orsak kan också vara det att optimeringen misslyckades eller att resultatet efter en körning inte överrensstämde med det

ursprungliga programmets resultatdata. Dessa versioner är dock inte på något sätt ”onödiga” eller överflödiga utan bidrar till optimeringen av programmet så länge introducerade buggar eller felantaganden korrigeras i efterliggande versioner.

Totalt har 38 versioner blivit skapade, av dessa 38 versioner finns mätningar från 30 versioner. De första versionerna 1-3 berörde endast omstrukturering av koden. Detta gjordes eftersom den ursprungliga versionen av Covana.cpp var skriven i endast en funktion. För att underlätta sökningen efter flaskhalsar ("*Hotspots*"), eller att söka efter den del av koden som tar mest tid under en körning, så delades programmet upp i tre nya funktioner. Efter uppdelningen bestod programmet av fyra stora funktioner:

- Main()
- ProbabilityCalc()
- MutualInformation()
- FindTriplet().

Alla testkörningar utfördes på sekvensfilen 7tm5.fasta. Programmets parametrar var följande:
Gap %: 0.21
Pvalue1: 0.04
Pvalue2: 0.20

4.1 Tidtagning och omstrukturering

Tidtagningen av programmet kallas för benchmarking på engelska. Detta utfördes genom att använda C++ bibliotekets time.h klass, samt att få totala körningstiden användes time kommandot i linux miljö. Ett annat benchmarking verktyg som användes var också profilerings kommando gprof vilket anger hur många anrop av funktioner och hur lång tid varje funktion tar. Ett problem som märktes i ett tidigt skede angående användningen av gprof var det att programmet var skrivet i en enda funktion main(). Detta gjorde att gprof inte var till stor nytta då main funktionen samt biblioteksfunktioner endast visade hur lång tid det tog.

Eftersom det ursprungliga Covana.cpp programmet innehöll endast en funktion Main() bestod det första steget att dela upp programmet. Efter att programmet studerats på bas av kommentarer som skrivits av Bairong Shen och den förklaring av programmets syfte som angetts delades programmet upp från en huvudfunktion till 4 huvudfunktioner, Main(), ProbabilityCalc(), MutualInformation() och FindTriplet(). Omstruktureringen tillåter profileringsverktyg som gprof att ge mer detaljerad information om var processorn spenderar mest tid.

I version Cov3 uppmärksammades två processorkrävande områden. Det första området befann sig i funktionen MutualInformation(). MutualInformation() funktionen räknar ut sannolikheten att aminosyra A och B befinner sig på positionerna i och j för varje position i aminosyresekvensen. Låt oss anta att man vid en sökning har 90 aminosyresekvenser eller proteiner av längden 1520 aminosyror, detta innebär att en slinga utför den ömsesidiga informationsuträkningen ca. 923 miljoner gånger. Orsaken är att den ömsesidiga informationsuträkningen utförs för varje aminosyra i varje sekvens. En jämförelse av alla positioner inom en sekvens kräver en dubbelnästad slinga såsom de två första slingorna i figur 15. Däremot de två innersta slingorna itererar max 400 gånger för varje length0 * length0 iteration, låt oss anta att length0 har värdet 1520, detta resulterar i 1520 * 1520 * 400 = 924 160 000 iterationer.

```
//length0 : Antal tecken i en sekvens.
for(i=0; i<length0; i++){
    for (j=i+1; j<length0; j++){
        for (ii=0;ii<20; ii++){
            for (jj=0;jj<20; jj++){
            ....
            }
        }
    }
}
```

Figur 15 Förenklat kodutdrag av den slinga som räknar ut ömsesidig informations.

4.2 Filhantering

Det tidigare kapitlet 4.1 beskriver antalet uträkningar som utförs för ömsesidig informationsuträkning. Det ursprungliga programmet utförde för varje möjlig kombination av aminosyreposition en filskrivning till hårddisken. Ifall längden på

antalet aminosyror i en sekvens är 1520, så utförs i värsta fall 2 310 400 stycken filskrivningar. För varje ömsesidigt informationspar, med ett värde över 0, som hittas kommer en ny fil att sparas till hårdskivan. Detta innebär i värsta fall ca. 2,3 miljoner filskrivningar per analys samt att lika många filer kommer att sparas på hårdskivan per analys.

Filskrivning i kombination med processor/minnes uträkningar är en väldigt tidskrävande process jämfört med att endast utföra RAM-minnesuträkningar, på grund av att en minnesåtkomst är mycket snabbare än en filoperation till en hårdskiva. Minnesoperationer kan vara upp till 1 000 000 gånger snabbare än filoperationer till en hårdskiva, se slutet av kapitel 2.3 för mera information. Ifall filskrivning utförs i samband med räkneoperationer så kommer processorns kapacitet inte att utnyttjas till fullo eftersom processorn måste vänta tills filoperationen är färdig förrän den kan fortsätta med nästa uträkning. Situationen med kombinerade RAM-minnesoperationer tillsammans med filoperationer kan optimeras om RAM-minnesoperationerna utförs först, och håller den informationen i RAM-minnet tills minnesuträkningarna är klara. Efter att minnesuträkningarna är färdiga kan de sparas ner till hårdskivan i en kontinuerlig diskoperation. Optimeringen kräver att problemet som behandlas inte kräver allt för mycket RAM-minne.

Det ursprungliga Covana programmet skriver data till ett antal olika filtyper. Filerna används senare av annan programvara för att representera informationen visuellt åt användaren. Totalt skrivs 16 olika filtyper ut av en körning, filinnehållet består av information om entropi, ömsesidiga informationspar som hittats samt alla tripletter som upptäckts.

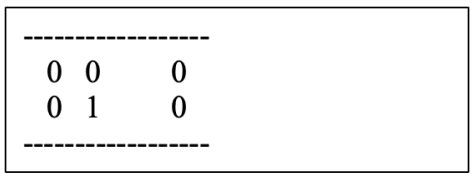
4.2.1 Optimering av filinnehåll

Optimering av ett program omfattar också att minska på användningen av hårdskiveutrymme. För att förbättra programmets prestanda behövs så litet information som möjligt lagras utan att informationen tappar värde. Covana programmets filer granskades för att se vilka optimeringar som kunde göras. Ömsesidiga informationsuträkningarna skapar filer av typen .pij. och

triplettanalysen skapar filer av filändelsen .tri. I programmets ursprungliga form visade det sig att antalet filer samt den information som filerna sparade innehöll överflödig information. Detta innebär att informationen kan representeras på ett effektivare sätt utan att förlora information.

De filer som skapades av programmet är menade att användas i andra sammanhang än i grundanalysen av aminosyresekvenser. Huvudsyftet med filerna, som skapas i analysen, är att på bästa sätt ge möjlighet att visualisera analysresultatet. Om strukturen på hur informationen representeras ändras, i optimeringssyfte, måste också de andra program som använder strukturen för visualiseringssyfte ändras.

Filtypen .pij innehåller information om det ömsesidiga informationsvärdet för aminosyrorna vid position i och j. En förbättringsmöjlighet upptäcktes här i tidigt skede genom att programmet skrev en .pij fil per godkänt ömsesidigt informationspar. I en körning med aminosyresekvensen Herpes_MCP kunde t.ex. 19083 ömsesidiga informationspar hittas. Det betyder att 19083 .pij filer skrivs ut. Innehållet i en .pij fil ser i den ursprungliga versionen ut på följande sätt:



Figur 16 Representation av .pij filinnehåll.

Representationen av .pij informationen ändrades och kommer att bli förklarad i närmare detalj i kapitel 4.8.2.

Triplettinformationen sparades i filer av formatet ”.tri”. Optimeringen angående filen förklaras i mera detalj i kapitlet 4.8.1.

4.3 Minnesanvändning

Det andra området som krävde mycket processor tid samt minne befann sig i FindTriplet() funktionen. En variabel kk användes för att initialisera en räcka muisite[][]. Variabeln kk beskriver antalet par aminosyror som har ett ömsesidigt

informationsvärde över en specifik gräns. Räckan muisite[][] initialiserades på följande sätt:

```
Int muisite[kk][kk];
```

I testfalls-körningen fick variabeln kk värdet 28441. Ett heltal använder sig av 4 byte för att representera sitt värde digitalt. Detta betyder att programmet reserverade 4*28441*28441 = 3,2 gigabyte minne.

Kk variabeln innehåller alla ömsesidiga informationspar som räknas ut. En viktig sak att komma ihåg är att alla ömsesidiga informationspar som identifieras antas ej till triplettuträkningen. Genom att vid en körning av programmet ha specifierat pvalue1 och pvalue2 argumenten påverkas korrelationsgraden mellan aminosyreparen vilket bestämmer de par som godtas till triplettuträkningen, (se kapitel 2.4 och 4.5)

4.4 Algoritmanalys av ömsesidig informationsuträkning

I processen att omstrukturera programmet identifierades två huvudsakliga flaskhalsar. Det första optimeringsförsöket gjordes genom att försöka minska på minneskraven för programmet. En orsak varför programmet kraschade var det att för mycket minne reserverades vid en analys. Ifall ett program reserverar lika mycket eller mera minne än det fysiska RAM-minne som en dator har så försöker operativsystemet först kompensera för bristen av fysiskt RAM-minne genom att använda sig av en swapfil på hårdskivan. En swapfil används som ett utökat RAM-minne så att datorn kan hantera situationen utan att krasha. Även om swapfilen används kan en reservation av för mycket minne innebära att datorn inte klarar av att hantera situationen och resulterar i en krasch. Eftersom operationer till swapfilen är mycket långsam jämfört med operationer till RAM-minnet bör swapfiler undvikas.

Efter att ha studerat programmet med hjälp av profileringsverktyget gprof samt att studera programmets kod hittades den största minnesanvändningen i FindTriplet() metoden. Den algoritm som det ursprungliga programmet använde sig av för att lösa triplettproblemet använde sig av en för stor minnesreserveation till

dubbelräckan eller matrisen "muisite". Deklarationen av räckan muisite[kk][kk] visade sig vara onödigt stor.

5	7	3	4	1	8	6	10	9	2
2	5		1	2		1	2	4	1
6	10		9	4		2	4	5	5
7			10	6		5	7	10	6
9							9		8
									10

Figur 17 Representation av den ursprungliga matrisen för FindTriplet() metoden.

Variabeln kk anger antalet par av aminosyror som har ett värde större än den fördefinierade gränsen. Räckan muisite[kk][kk] initialiserades för att kunna representera alla aminosyrepars positioner som har identifierats från MutualInformation() metoden. Genom att se på representationen går det att se att det går att förenkla. Se figur 17.

1	2	3	4	5	6	7	8	9	10
2	1		1	2	1	5		4	2
4	5		9	6	2	10		5	4
6	6		10	7	5			10	7
	8			9					9
	10								

Figur 18 Sorterad representation av matrisen för FindTriplet() metoden.

X-axeln representerar en aminosyreposition i och värdet på Y-axelns led är position j. X-axelns storlek beror på aminosyresekvensens längd. Det kan teoretiskt finnas lika många fyllda platser på Y-axeln som aminosyresekvensens längd minus ett, eftersom det inte går att relatera samma position i sekvensen med sig själv. Det går också att se att eftersom matrisen representerar indexet eller positionen för en viss aminosyra på X-axeln så behöver matrisen inte initialiseras på bas av variabeln kk, utan antalet aminosyror i sekvensen istället, dvs. variabeln resnum kan användas.

Att konstatera detta är en optimering som minskar mängden alternativ som behöver tas i beaktande för korrelationsanalys. I exempelkörningen har variabeln kk värdet 28441 och variabeln resnum har värdet 1520.

Däremot är behovet av antal platser på y axeln okänd förrän aminosyresekvensen har analyserats färdigt. Dvs. det går inte att på förhand bestämma hur många aminosyror som är relaterade till aminosyran på position 1. Därför kan också storleken ändras för Y-axeln i matrisen. Y-axelns storlek ändrades därför till det teoretiska maximiantalet par som i teorin kan relateras på bas av aminosyresekvensens längd, se figur 19 nedan.

```
N=length0*(length0-1)/2;
```

Figur 19 Length0 innehåller värdet av sekvensens längd.

Figur 17 visar den ursprungliga representationen av muisite matrisen. Den initialiserades på bas av kk variabeln genom: muisite[kk][kk]. Representationen sparar båda referenserna från aminosyra A till aminosyra B och tvärtom från aminosyra B till aminosyra A. Det betyder att dubbelinformation finns angående en relation av aminosyreparet. Funktionen som utför ömsesidiga informationsuträkningar beskrivs i kapitel 4.5 på det sätt hur informationen generellt används för att identifiera tripletter. I kapitel 4.5.4 beskrivs hur jag introducerar en optimerad datastruktur för triplettrepresentation, som minskar på kravet av hårdskiveutrymme samt möjliggör att snabba upp algoritmen som identifierar tripletter med hjälp av mera avancerade sökalgoritmer.

4.5 Optimering av algoritmen findTriplet

Syftet med metoden findTriplet() är att se ifall tre stycken aminosyror i sekvenserna är relaterade. För att förstå hur implementationen optimerats behöver vi gå in på algoritmens syfte. Vad är en triplett och hur identifieras den?

Låt oss säga att vi har tre stycken aminosyror: A, B och C. De befinner sig respektive på positionerna 1, 2 och 6 i aminosyresekvensen. Genom att se på figur 18 så ses att aminosyra A i position 1 innehåller tre relationer 2, 4 och 6. Det här betyder att aminosyra A är relaterad med aminosyrorna på positionerna 2, 4 och 6.

Nästa steg för att identifiera en triplett börjar med att algoritmen går till följande relaterade aminosyreposition för att se om någon annan aminosyra som är relaterad med aminosyra A också är relaterad med aminosyra B. Det vill säga aminosyra A's relationer inspekteras.

Eftersom aminosyra A på position 1 är relaterad med aminosyra B på position 2 ser vi närmare på besläktade aminosyror i kolumn 2 som nästa. Aminosyra B är enligt figur 18 relaterad med aminosyrorna i positionerna 1, 5, 6, 8 och 10. För att nu kunna konstatera om en aminosyra C är relaterad med både A och B så jämförs de resterande relationerna för aminosyra A och B för att se om de matchar. Genom att söka vidare i kolumn 1 och 2 så går det att se att position 4 inte finns i kolumn 2 men däremot så finns nummer 6 i både kolumn 1 och 2. Detta betyder att nästa steg är att se i kolumn 6 efter att båda positionerna 1 och 2 finns i dess kolumn. Detta betyder att aminosyra A och B är relaterad med varandra samt att A och C är relaterad och att B och C är relaterad med varandra. Det här innebär att en triplett med aminosyror på platserna (1,2,6) har hittats.

4.5.1 Identifierade optimeringar

Detta utförs för varje rad och varje kolumn i matrisen. Den ursprungliga implementationen av triplettsökningen utförde en koll per cell i hela matrisen oavsett om det fanns något värde sparat i matrisen eller inte. Matrisen var inte sorterad utan informationen var insatt i den ordning som från algoritmen ansågs vara godtycklig. Matrisen reserverade onödigt mycket utrymme för att spara ömsesidig informationsparen. Alla dessa orsaker bidrog till att programmet tog onödigt länge att köra samt att allt för mycket minne blev reserverat för programmets exekvering.

4.5.2 Optimering av identifierade ”Hot spots” i metoden FindTriplet().

För att tackla problemet med den stora minnesallokeringen i samband med triplettsökningen så introducerades en ny datastruktur i version 5 av covana. Datastrukturen består av två stycken integers och en double datatyp. I dessa datatyper sparas positionerna för de relaterade aminosyrorna och de ömsesidiga

informationsvärdena. I koden ser deklarationen av datastrukturen ut på följande sätt:

```
struct Mutobj {
  double muinf12;
  int site1;
  int site2;
};

n=length0*(length0-1)/2;
  //the theoretical maximum number of mutual information
pairs that may be calculated and stored in memory.

Mutobj *mo = (Mutobj*)malloc(sizeof(Mutobj)*n);
```

Figur 20 Kodutdrag, MutObj strukturen.

Mutobj strukturen ersatte i det ursprungliga programmet tre räckor. De tre räckorna motsvarar muinf12, site1 och site2. Genom att ändra dessa tre räckor till en gemensam struktur fås lättare en överblick av koden. Det är också enklare att programmera när data grupperas ihop till mindre objekt.

Det ursprungliga programmets ömsesidiga informationsalgoritm utförde en sortering på bas av det ömsesidiga informationsvärdet i alla aminosyrepar som identifierades. Märk väl att de ömsesidiga informationsparen inte var sorterade på bas av positionernas platser. Detta betyder att värdena på X-axeln i Figur 17 inte var sorterade ännu i denna version såsom det går att se att de är i Figur 18.

Sorteringen utfördes mha. bubbelsorteringsalgoritmen, denna algoritm blev utbytt mot Quicksortalgoritmen det vill säga komplexitet från bubbelsorteringens $O(n^{\wedge 2})$ till Quicksortalgoritmens nLog(n). Quicksortalgoritmen implementerades mha. en standard biblioteks Quicksortfunktion i C++ [9].

Programmets totalkörtid förbättrades från 227,83 sekunder från version 1 till 202,67 sekunder i version 4. Minnesanvändningen minskades från 3250 Mb till 110 Mb, detta resulterade i en förbättring med ca 15 sekunder. Lagringsutrymmet på hårdskivan för en körning förbättrades inte i denna optimering.

4.5.3 Optimering av triplettsökningsalgoritm

I version 6 blev funktionen FindTriplet() optimerad. Efter att koden studerats lades det märke till att triplettalgoritmen utförde onödiga uträkningar. Triplettalgoritmen består av fyra stycken slingor. Slingorna går igenom varje cell inom matrisen på det sätt som förklarades i föregående kapitel. I och med att informationen som sparas i matrisen inte är sorterad är det svårt att införa kontroller som förutser ifall en nyckel finns före eller efter en viss position. Däremot om informationen är sorterad går det lättare att kontrollera ifall en viss kolumn kommer att innehålla förväntade värden och på bas av detta antingen avbryta eller fortsätta sökningen. Genom att se på det ursprungliga programmets minneskrav på ca. 3 Gb går det att förstå att triplettalgoritmen tar lång tid.

I denna version infördes kontroller som ser efter ifall den första cellen i Y-axelns led är tom, om så är fallet avslutas slingan och fortsätter med nästa kolumn. Detta minskar på antalet onödiga operationer.

4.5.4 Optimering av triplettalgoritmens minnesanvändning

I version 5 minskades minnesanvändningen med hjälp av att definiera om storleken på de räckor som innehåller positionerna på de aminosyrepar samt de ömsesidiga informationsvärdena som ska användas i matrisen muisite, se kapitel 4.5.2. När det konstaterats att optimeringen fungerade och att den producerar samma resultat som det ursprungliga programmet så går det att fortsätta att se om minnesanvändningen kan minskas.

Genom att se på datastrukturen som introducerades i version 5 går det att se att den använder sig av två stycken integer datatyper. En integer representeras av 4 bytes. Eftersom vi endast vill visa positiva värden dvs., indexpositioner för aminosyror, går det att byta ut integer datatypen mot en short. En short representeras av 2 bytes vilket betyder att minnesanvändningen ännu minskar.

Datastrukturen som introducerades i version 5 används i funktionen MutualInformation() för att lagra och sortera alla aminosyrepar. Datastrukturen

gjorde också så att sorteringen lättare kunde användas av Quicksortalgoritmen. Åtgärden var det andra steget i optimeringen av triplettsökningsalgoritmen.

I version 7 introducerades en ny datastruktur istället för den ursprungliga MuiSite matrisen. Till strukturen kopieras de aminosyrepar, som har tillräckligt högt ömsesidigt informationsvärde, från den tidigare MutObj strukturen ovan. Tidigare sparades informationen i dubbel räckan muisite som representeras nedan i figur 21.

```
  int** muisite;
  //a 2D array for store the covariant positions (site1, site2)
  cout << "kk @ muisite: " << kk;
  try {
    muisite = new int*[kk];
    for (k=0; k<kk; k++)
    muisite[k]= new int[kk];
  } catch (...) {
    cout<<"Exception raised: muisite"<<"\n";
    return 1;
```

Figur 21. Instantiering av MuiSite matrisen i det ursprungliga programmet.

Den nya datastrukturen definierades på följande sätt i figur 22.

```
struct MuiSite {
  int sitei;
  int sizeofj;
  int *sitej;
  int occupiedj;
};
```

Figur 22. Instantiering av MuiSite matrisen i det ursprungliga programmet.

Och initialiserades på följande vis:

```
MuiSite *ms;
ms = (MuiSite*) malloc(sizeof(struct MuiSite)*topnum);
initms(ms, topnum);

...

int initms(struct MuiSite *mss, int length) {
  MuiSite *ms = (struct MuiSite*)mss;
  for (int i = 0; i<length; i++) {
    ms[i].occupiedj=0;
    ms[i].sitei=0;
    ms[i].sizeofj=0;
    ms[i].sitej = (int *)malloc(sizeof(int));
  }
}
```

Figur 23 Initialisering av MuiSite strukturen.

En skilld metod för att initialisera matrisen ms skapades. Metoden nollställer samtliga datatyper och allokerar minne för räckan sitej. Då ett nytt par identifierade aminosyror ska läggas till används en metod addms():

```
int addms(struct MuiSite *mss, int index, int site1, int site2) {
  /*
    .sizeofj = Number of elements in the array sitej.
    .occupiedj = Current size of the array sitej.
   */
  int retval=0;
  int tmp=0;
  try {
    MuiSite *ms = (struct MuiSite*)mss;
    if (ms[index].sizeofj>=ms[index].occupiedj) {
      ms[index].sitej = (int*)realloc(ms[index].sitej,
sizeof(int)*(ms[index].occupiedj+32));
      ms[index].occupiedj += 32;
    }
    ms[index].sitei = site1;
    ms[index].sitej[ms[index].sizeofj] = site2;
    ms[index].sizeofj++;
    tmp = ms[index].sizeofj;
  } catch (...) {
    cout << endl << "Failed to add element.";
    exit(1);
  }

  return retval;
}
```

Figur 24 Metod för att registrera ett par aminosyror till matrisen MuiSite.

Det vill säga funktionen addms tar räckan med datastrukturen MuiSite som argument och de positions Den första aminosyrans position tilldelas integern sitei. Den andra positionen läggs till i räckan sitej. När värdet av den andra positionen sätts in i räckan sitej så reallokeras minnet på räckan sitej. Därefter uppdateras information om hur många element som finns i räckan sitej samt hur mycket minne som den använder för tillfället.

Efter att alla par av ömsesidigt informationsvärde har blivit tillagda i matrisen MuiSite så sorteras matrisen på bas av sitei. Dvs, en sortering av alla element i räckan så att den får informationen sparad i stigande ordning på bas av den första aminosyrans position. Detta resulterar i en liknande matris som det går att se i figur 25. I figur 25 beskrivs hur X-axelns värden går i stigande ordningsföljd.

Genom att sortera matrisen på detta sätt fås tillgång till mer effektiva sökningsalgoritmer. Vetskapen om att endast stigande värden kommer att finnas när algoritmen går vidare till nästa kolumn är också garanterad. Genom att jämföra

med den ursprungliga triplettalgoritmen går det nu att avskaffa en del villkorssatser och på så vis också vinna processortid.

Den största ändringen gällande matrisen, innehållande datastrukturer av typen MuiSite, var att den nu minskade i antalet element som behöver användas för att representera samma information som tidigare. Ändringen minskar på behovet av minnesutrymme med mera än hälften. Tidigare sparades båda positionerna för en relation mellan ett par aminosyror. Ifall aminosyra A är relaterad med aminosyra B så betyder detta också att B är relaterad med A. Med hjälp av denna information behöver inte relationen B->A sparas ner, utan endast A->B, se figur 25.

1	2	3	4	5	6	7	8	9	10
2	5		9	6		10		10	
4	6		10	7					
6	8			9					
	10								

Figur 25. Exempel på hur informationen är sorterad i stigande ordning i X-axelns riktning i matrisen MuiSite.

Det går också att se att antalet värden som sparas ner i kolumnerna kan vara [längden av sekvensen] – [indexpositionen] i respektive kolumn. Detta betyder att ett teoretiskt maximiantal relationer kan definieras och kan liknas vid en matris så som illustrationen i figur 26.

x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	
x	x	x	x	x	x	x	x		
x	x	x	x	x	x	x			
x	x	x	x	x	x				
x	x	x	x	x					
x	x	x	x						
x	x	x							
x	x								
x									

Figur 26. Illustration av ett teoretiskt maximiantal element i matrisen MuiSite.

Ovanstående antaganden möjliggör ännu en minskning på allokeringen av minnet för matrisen MuiSite.

När ändringarna blev implementerade sattes också en kontroll in i triplettalgoritmen att ifall en cell i räckan sitej[] är tom så går det att avbryta sökningen i den räckan. Det går att göra eftersom addms funktionen endast fyller på värden genom att reallokera minne för nästa värde, den sätter inte in värden i en fördefinierad räcka.

Det ursprungliga programmets triplettalgoritm utförde läs och skrivoperationer i filsystemet direkt när en triplett hade identifierats. Filoperationen gjordes i samband med uträkningar inuti en fjärde nivåns slinga. Att göra en filoperation i samband med snabba minnesuträkningar gör att körningen kan vara mycket långsam. Tiden det tar att göra en hämtningsoperation från RAM-minne mäts i nanosekunder, jämfört med en hämtningsoperation från t.ex. en hårdskiva så mäts tiden i millisekunder.

För att åtgärda detta skapades en ny datastruktur för att hålla information om identifierade tripletter. Då en triplett blir identifierad så sparas de tre positionerna inom tripletten ned till en räcka med strukturen:

```
struct Triplet {
  int s1;
  int s2;
  int s3;
};
```

Figur 27 Datastruktur som beskriver triplettinformation

Då triplettalgoritmen avslutats så finns all triplettinformation i räckan:

```
Triplet *tp;
tp = (Triplet*) malloc(sizeof(Triplet)*tpsize);
```

Figur 28 Exempel på hur en datastruktur med triplettinformation

Nu går det att i efterhand skriva all triplettinformation på en och samma gång. Istället för att göra detta inuti slingan.

För att hitta en triplett i den nya representationen av matrisen muisite så kan proceduren vara följande:

1) Ta reda på vilka värden som finns i kolumn 1.

2) Jämföra ifall någon av dessa värden existerar i någon av de andra kolumnerna.
3) Ifall en match hittas betyder det att en triplett identifierats.

Med den nya algoritmen behövs inte lika många kontroller som i den ursprungliga triplettsökningsalgoritmen.

Effektivitetsmässigt var optimeringarna från version 6 till version 7 de mest givande. Körningstiden för programmet minskade från 216 sekunder till 49,4 sekunder och minnesanvändningen minskade från 110 Mb till 68 Mb.

4.6 Optimering av den ömsesidiga informationsdatastrukturen

När den nya representationen av matrisen MuiSite införs leder det till att ytterligare andra metoder kan användas för att optimera ömsesidiga informationsalgoritmen vidare, detta sker i version 8. Det värde som hittills använts för att initialisera matrisen MuiSite består av värdet i variabeln topnum. Variabeln topnum beskriver alla ömsesidiga informationspar som hittats och i testfallet var värdet för variabeln topnum 1669. Detta värde är för stort för att allokera den exakta mängden minne för matrisen MuiSite. Genom att ta figur 25 till hjälp så ser vi att värdena på första raden representerar positionerna för sitei, dvs. positionen för den första aminosyran i ett ömsesidigt informationspar. Det betyder att varje aminosyreposition skall ha sin egen kolumn i matrisen och efter det behövs inga mera kolumner. Antalet värden som kommer att läggas till i kolumnerna är okänt men vore bra att veta i det skede då minnet allokeras. Eftersom sitej representerar en aminosyras relationer till andra aminosyror och i denna del av programmet är antalet relationer ännu okänt, härmed utförs minnesallokeringen dynamiskt. Detta betyder att under körningens lopp kommer räckans minneskrav att öka för att få plats med de värden som behöver sparas i räckan.

Genom att reservera minne efter behov kan RAM-minnet fragmenteras. Detta innebär att minnesplatsen för räckan inte kommer i en kontinuerlig följd i minnet. För att förhindra minnesfragmentering är det bra att veta hur mycket minne som kommer att krävas. För att få reda på mängden minne som behövs går det att

uppskatta minnesbehovet på bas av ett scenario som beskriver det värsta fallet. Denna uppskattning anger minnesbehovet ett värde som kan användas för att kontinuerligt allokera minnet, även om man allokerar onödigt mycket minne. När insättningen av data är färdig går det att i efterhand att minska på datastrukturens storlek till den exakta mängden efter behov. Ändringen gör att minnesanvändningen minskade i metoden findTriplet().

4.7 Binärsökningsalgoritm

En optimering i metoden findTriplet() berör binärsökningsalgoritmen. Triplettsökningsalgoritmen i det ursprungliga programmet använde sig av linjärsökning (Linear Search på engelska). Det betyder att algoritmen går igenom cell efter cell i varje kolumn för att se om den hittar det värde som söks. Tidsestimeringen av en körning med sekventiellsökning är O(N). Sökningsmetoden var lämpad i det ursprungliga programmet eftersom informationen inte var sorterad samt att den inte är svår att implementera.

Då all data i matrisen är sorterad och tillgänglig går det att använda sig av mer effektiva sökningsalgoritmer. Den algoritm som valdes i denna optimering är binärsökning (Binary Search på engelska). Binärsökning är en rekursiv algoritm som använder söndra och härska-strategin. Algoritmen kräver att en sorterad räcka används. Binärsökning delar upp sökområdet i flera delar, varje del behandlas som ett nytt sökområde. Delarna halveras och i varje del kontrolleras om nyckeln finns till vänster eller höger av delningspunkten. Om nyckelns värde befinner sig till vänster om hävstångspunkten (Pivot) så fortsätter algoritmen att dela upp den vänstra halvan i hälften för att fortsätta på samma sätt. Algoritmen har en tidsestimering på $Log_2(N)$ [3].

Effektivitetsmässigt snabbade ändringen upp triplettsökningen med ungefär 5 sekunder från version 8 med 51,7 s till version 9 med 46,8 s.

4.8.1 Komprimering av data i triplettfiler

I version 10 optimerades filinnehållet i triplettfiler. Det antal filer som programmet skrev till hårdskivan var ett stort problem eftersom informationen som skrevs ner till filer innehöll mycket redundant information. Funktionen findTriplet() skrev information angående identifierade tripletter till två filer med formatet ”.tri.cov” och ”.tri”. Filerna innehöll båda samma information angående de aminosyrors relationer som formar tripletter. Det som skiljde filerna åt var det att ”.tri.cov” filen var formaterad för att läsas upp i html kod. Den innehöll formaterings information hur texter skall visas i ett annat program.

Ett exempel av en post i en fil med typen .tri:

```
   "D10" ->   "V11"[style=bold,color=blue];
   "V11" ->   "N176"[style=bold,color=blue];
   "N176" ->   "D10"[style=bold,color=blue];
//   1:
```

Figur 29 Exempel på filinnehåll i triplettfiler (.tri) i den ursprungliga versionen.

Det går att se att aminosyra D på plats 10 är relaterad med aminosyra V på plats 11 och V med N samt N med D vilket visar alla tre relationer för en triplett. Det går också att se att onödig information sparas per triplett. T.ex. så skrivs formaterings text som senare kan användas av en parsnings funktion som representerar informationen. Eftersom en körning genererar ett stort antal tripletter så kan formateringstexten tas bort så att antalet tecken som behövs minimeras för att spara den information som senare behöver visas åt en användare.

Det går att konstatera att exemplet ovan inte är optimalt för att representera en triplett. Ett försök att minimera antal tecken för att representera en triplett gjordes och följande format användes istället:

```
1:D10,V11,N176
```

Figur 30 Innehålletet på .tri filinformation efter optimering.

Det nya formatet representerar vilka positioner och vilka aminosyror som utformar en triplett på ett enkelt sätt. Om det gamla och nya formatet jämförs i ett exempel

går det lättare att förstå hur mycket mindre lagringsutrymme som behövs som ett resultat av optimeringen.

Låt oss anta ett exempel där en aminosyresekvens innehåller 1520 tecken och i sekvensen hittas 627397 tripletter. Det gamla filformatet behövde 171 – 180 tecken för att spara ner information om en triplett. Det nya formatet behöver endast 10 – 19 tecken per triplett. Antalet tecken beror på hur många tecken som behövs för att representera aminosyrans position. Storleksskillnaden i detta exempel blir då:

Gamla formatet: 627397*180 = 112,9 Mb

Nya formatet: 627397*19 = 11,9 Mb.

Genom att använda det nya formatet minskar filutrymmet för triplettinformationen med ungefär 10 gånger.

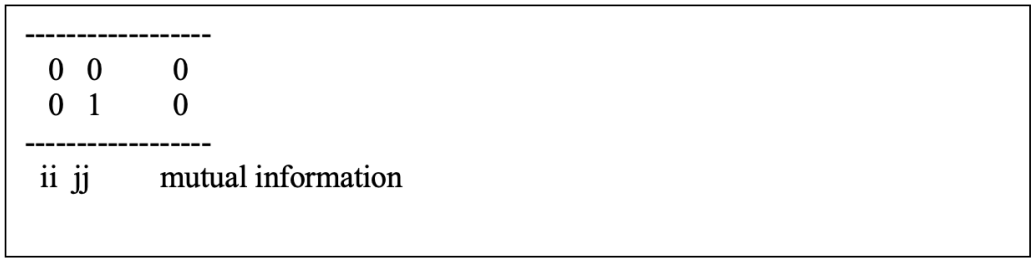
Denna ändring kräver att det program som använder ”.tri” filer också behöver anpassas. Programmet måste se till att sköta om formateringen på texten. Det är bättre att låta ett externt program sköta om formateringen istället för att spara ner hur formateringen skall se ut för varje triplett med tanke på lagringsutrymme och processortid.

4.8.2 Komprimering av data i filer som representerar ömsesidig information

Efter den lyckade optimeringen av triplettfilskrivningen koncentrerades nästa optimering på ”.pij” filskrivningen. Filskrivningen var en av orsakerna varför programmet var väldigt tungt och tidskrävande. Funktionen MutualInformation() söker efter aminosyrepar som är relaterade. Informationen om varje par som identifierats som ett ömsesidigt informationspar sparas ner till en fil. Det ursprungliga Covana programmet skapade en fil innehållande det ömsesidiga informationsparets information för varje par. I en körning med kommandot:

./covana Herpes_MCP 0.21 0.04 0.2

så hittades 19083 antal ömsesidiga informationspar. Det betyder att programmet skapade 19083 stycken filer. Detta är ett tungt jobb för hårdskivan samt filsystemet med tanke på att analysen antagligen ska köras flera gånger. I figur 31 går det att se ett exempel på hur informationen representerades i en .pij fil:



Figur 31 Exempel på den ursprungliga strukturen på en post i .pij filer.

Den gamla representationen innehöll många noll tecken. Den egentliga informationen omringades av många överflödiga tecken. En orsak varför programmet designats på detta sätt kan vara att det externa program som skall läsa filen kräver att filinformationen är strukturerad på detta sätt. I optimeringssyfte är det skäl att ändra på strukturen så att så mycket information som möjligt kan representeras med så få tecken som möjligt. Denna typ av ändring sparar hårdskiveutrymme. Programmet som ska tolka filens innehåll i efterhand behöver också ändras för att förstå sig på informationen.

Filformatet komprimerades till följande format, se figur 32.

Figur 32 Exempel på optimerad datastruktur för ömsesidig information.

Den ursprungliga versionen av programmet skapade en ny fil per identifierat ömsesidigt informationspar. Att skapa filer är tidskrävande och en optimering kan vara att undvika filoperationer som berör skapande av nya filer samt att minska på antalet skrivoperationer till filsystemet. Den nya optimeringen består av att spara all nödvändig information till en och samma fil för alla identifierade par. Detta kräver att det andra programmet som vill läsa den nya filen behöver ta ändringen i beaktande, den visualiserande delen av ProCon tas inte upp i detta diplomarbete.

4.8.3 Förbättring av filskrivningsalgoritm

Ändring till filskrivningsalgoritmen gav resultat som kan ses i version 12 och 13 av covana.cpp programmet, här förbättras den metod som används för att skriva information till filer. Den ursprungliga versionen av programmet använde kod beskriven i Figur 33:

```
ofstream file;
char testoutfile[40];
testtriplet.open(testoutfile);
strcpy(testoutfile, argv1);
strcat(testoutfile, ".tst");
testtriplet.open(testoutfile);
testtriplet << "Denna text sparas I filen.";
testtriplet.close();
```

Figur 33 Exempel på det ursprungliga programmets filkrivningsförfarande.

För varje text som blev skriven till filen så krävs en filhämtningsoperation. Ifall filskrivningsoperationer utförs inuti slingor där andra uträkningar utförs samtidigt, kommer processorns maximala kapacitet inte till full användning. Detta beror på att operationer till filsystemet är mycket långsammare än RAM-minnesoperationer.

Genom att använda en funktion som stringstream för att skriva till filen så cacheas skrivningen och allt går mycket snabbare. Genom att spara den information som behövs till filen i en stringstream så kan filskrivningen skötas på en gång. Att skriva till hårdskivan all den information som finns i stringstreamen på en gång, i stället för att synkront uppehålla processorn för att vänta tills flera små deloperationer till filsystemet måste utföras, betyder att processen blir snabbare. Att använda stringstream i samband med filskrivningen snabbades programmet upp med ca. 3 sekunder från föregående version.

Efter att version 12 förbättrade programmet med hjälp av stringstream, som konstaterades att snabba upp filskrivningen, blev nästa steg att implementera metoden att skriva filer i resten av programmets filskrivningsprocedurer. Programmet snabbades upp med 0,25 sekunder på grund av ändringen.

4.9 Onödig slinga i sortering av element i MS-matrisen

I denna optimering togs en felaktig slinga bort. Slingan tillhörde en sortering av sitej[] räckan eller en sortering av värdena i kolumnerna i matrisen muisite. Slingan kan anses vara ett programmeringsmisstag, se figur 34 nedan.

```
 //Sorting ms on key sitei.
qsort(ms,length0,sizeof(struct MuiSite), cmp_MuiSite);
int sj=0;
for (i=0; i<length0; i++) {
  sj = ms[i].sizeofj;
  for (j=0; j<sj; j++) {
    qsort(ms[i].sitej,sj, sizeof(int), cmp_integer);
  }
}
```

Figur 34 Sortering av värden i matrisen ms - onödigt många sorteringar.

Sorteringen av räckan sitej[] behöver endast utföras för så många sitei värden som finns i räckan ms. Den slinga som togs bort ändrade koden att se ut enligt figur 35:

```
//Sorting ms on key sitei.
qsort(ms,length0,sizeof(struct MuiSite), cmp_MuiSite);
int sj=0;
for (i=0; i<length0; i++) {
  sj = ms[i].sizeofj;
  qsort(ms[i].sitej,sj, sizeof(int), cmp_integer);
}
```

Figur 35 Sortering av matrisen ms, optimerad version.

Integer datatyper som endast representerar positiva heltalsvärden blev omdefinierade till unsigned integerdatatyper. Det betyder att om unsigned integer datatyper används vinns 2 byte per datatyp. Ett exempel av strukturen MuiSite som representerar den ursprungliga problem matrisen kan ses i koden nedanför:
Se figur 36 hur den ursprungliga koden ändrades från:

```
struct MuiSite {
  int sitei;
  int sizeofj;
  int *sitej;
  int occupiedj;
};
```

Figur 36 MuiSite struktur före optimering av datatyp.

Se figur 37 hur koden optimerades:

```
struct MuiSite {
  unsigned int sitei;
  unsigned int sizeofj;
  int *sitej;
  unsigned int occupiedj;
};
```

Figur 37 MuiSite struktur med optimerade datatyper

4.10 Triplettalgoritm med binärsökning

I version 15 förbättrades triplettsökningsalgoritmen. Då en sökning utförts i en kolumn på bas av en nyckel, så finns redan information om var algoritmen ska fortsätta sökningen. Denna information användes inte i den ursprungliga triplettsökningsalgoritmen. Den ursprungliga sökningsalgoritmen gick igenom och jämförde alla element i räckan. Så som i kodutdrag figur 38:

```
for (i=0; i<length0; i++) {
    if (ms[i].occupiedj == 0) continue;
    for (j=i+1; j<length0; j++) {
      for (k=0; k<ms[i].sizeofj; k++) {
        if (ms[i].sitej[k] == ms[j].sitei) {
          for (l=k+1; l<ms[i].sizeofj; l++) {
            for (m=0; m<ms[j].sizeofj; m++) {
              if (ms[i].sitej[l] == ms[j].sitej[m]) {
                if (numoftriplets >= (tpsize)) {
              tpsize = tpsize + tpincrementsize;
              tp = (Triplet *)realloc(tp,sizeof(Triplet)*(tpsize));
                  cout << endl << "reallocating to: " <<
sizeof(Triplet)*(tpsize);
              }
              tp[numoftriplets].s1 = ms[i].sitei;
              tp[numoftriplets].s2 = ms[j].sitei;
              tp[numoftriplets].s3 = ms[j].sitej[m];
              numoftriplets++;
                }
```

Figur 38 Kodutdrag findTriplet algoritm utan binärsökning.

Algoritmen arbetade ursprungligen med osorterade värden. I den optimerade versionen används sorterade data och kan därför utnyttja binärsökningsalgoritmen i algoritmen findTriplet, se figur 39.

```
for (i=0; i<length0; i++) {
    if (ms[i].occupiedj == 0) continue;
    for (j=i+1; j<length0; j++) {
      for (k=0; k<ms[i].sizeofj; k++) {
        if (ms[i].sitej[k] == ms[j].sitei) {
          for (l=k+1; l<ms[i].sizeofj; l++) {
            key = ms[i].sitej[l];
            element    =    (int*)bsearch(&key,    ms[j].sitej,    ms[j].sizeofj,
sizeof(int), cmp_integer);
if (element != NULL) {
  ...
  numoftriplets++;
```

Figur 39 Exempel av en del av findtriplet algoritmen med binärsökning.

Denna implementation sparar antalet sökningar i matrisen muisite. Förbättringen minskade körtiden med 0,92 sekunder från 3,64 sekunder till 2,72 sekunder. Förbättringen gjorde programmet 25% snabbare jämfört med version 14.

4.11 Implementation av uppslagstabell

Funktionen ProbabilityCalc() utför en analys av aminosyrorna genom att räkna var och en unik aminosyra samt antalet aminosyror per grupp. Detta görs genom att använda en matris av storleken 20x20, eftersom det finns 20 stycken olika aminosyror. Indexena pekar på var och en unik aminosyra samt hur många av de andra aminosyrorna den är relaterad till. Den ursprungliga implementationen bestod av två stycken switchar med 20 stycken fall var.

4.12 Omordning av fall i switch sats enligt mest frekventa fall

Omordning av fall i switch-sats, optimeringsmetoden beskrivs i optimeringsguiden [11, s.28]. I början av Covana koden finns en funktion där aminosyror grupperas från ”.fasta” filen. Funktionen flyttades in i den nya metoden ProbabilityCalc(). Här används en switch sats för att på bas av tecken kunna använda informationen i analysen. Meningen med switch satsen är att för varje aminosyra registrera vilken aminosyra som förekommer samt registrera till vilken grupp den hör. I switchar ska, i optimeringssyfte, eftersträvas att få det mest förekomna fallet på första plats och de mer sällan förekomna fallena på senare platser [11, s. 28]. En switch fungerar så att processorn börjar med att kontrollera det första fallet, sedan nästa osv. ända

tills det sista fallet. I genetiken har studier gjorts så att de aminosyror som är statistiskt mest förekommande har identifierats. Detta har använts för att ordna switch fallen i den optimerade ordningen. Enligt webreferensen [12] så beskrivs förekomsten av olika aminosyror.

4.13 Allokeringen av minnet ändrades från gles till kompakt

I version 18 ändrades minnesallokeringen från distribuerad till kompakt. Optimeringen beskrivs i optimerings guiden [11, s. 19]. Ändringen berör reservation av minnet för matrisen seq:

```
char** seq;
 //used for residues and sequences, 2D array: (sequence + position)
 cout<<"....."<<endl;
 try {
  seq = new char*[MaxSeqN];
  for (k=0; k<MaxSeqN; k++)    seq[k]= new char[MaxResN];
 } catch (...) {
  cout<<"Exception raised: for seq"<<"\n";
  return 1;
 }


 // initializing seq
 for (k=0; k<MaxSeqN; k++)  *seq[k]=0;
```

Figur 40 Ursprungliga minnesallokeringen av seq matrisen.

```
char **seq = (char**)malloc(rows*sizeof(char*));
 char *SEQ = (char*)malloc(rows*columns*sizeof(char));
 for (i=0; i<rows; i++) {
  seq[i] = SEQ+ i*columns;
  for (j=0; j<columns; j++) {
            seq[i][j] = ' '; //Initialization.
  }
 }
```

Figur 41 Exempel på kompakt minnesallokering.

Ändringen påverkade körtiden från 11.98 sekunder till 11.54 sekunder.

4.14 Ersättning av uträkningar med konstanter

Redan uträknade värden sparades ner till variabler i första uträkningen och efter det användes variablerna istället för att räkna ut samma sak igen. Detta minskar på antalet instruktioner som processorn behöver utföra. Ett exempel på detta är då värdet av log(20.0) räknades ut. Samma uträkning utfördes i en sekvens med 1520 tecken 1520 gånger. Optimeringen i detta fall är att flytta ut uträkningen och spara den i en variabel, eller att sätta in en konstant med tillräckligt många decimaler. Denna optimering är något som dagens moderna kompilatorer troligen kan hantera automatiskt.

4.15 Exponentsökningsalgoritm kombinerad med binärsökning

Triplettalgoritmen bestod av linjärsökning, dvs varje element i matrisen jämfördes för att hitta en matchning. I version 21 av covana introducerades exponentsökningsalgoritmen. Redan i version 9 som beskrivs i kapitel 4.7 introducerades binärsökning istället för sekventiellsökning. Denna version tar ett steg längre för att undersöka om en ännu snabbare triplettsökningsalgoritm kan åstadkommas.

Exponentiellsökningsalgoritmen implementerades så att ifall den hittar en nyckel kommer värdet att returneras. Om den inte hittar ett värde kommer den att jämföra nyckel värdet med de interval den söker igenom, ifall nyckeln är mindre än nästa intervall så kommer den att returnera tidigare index samt nästa index. Dessa två index värden används vidare av binärsökningsalgoritmen för att granska det angivna området i detalj.

4.16.1 Uppveckling av slinga

Uppveckling av slinga (Loop unrolling på engelska) beskrivs i "*Software optimization Guide"* [11, s. 13]. Detta innebär att små nästade slingor som utför t.ex. 2 iterationer inne i en större slinga kan skrivas om så att den lilla slingan ersätts med direkta referenser till de element som slingan itererar över. Enligt "Software

optimization Guide" [11, s. 13] minskas antalet instruktioner som utförs i processorn.

Genom att explicit utföra fler uträkningar per iteration istället för att göra en uträkning per iteration kan också öka på antalet parallellt exekverbara instruktioner[11, s.35-36. Detta beskrivs i optimeringsguiden för AMD-athlon processorer [11, s. 35-36]. Tekniken används i metoden som räknar ut ömsesidig information, se exempel på uppveckling av slinga samt explicit uträkning för att öka parallellism i Covana koden nedan:

```
for(i=0; i<kk; i++) {
    standdev = standdev + (mo[i].muinf12- avemuinf)*(mo[i].muinf12-
avemuinf)/(kk-1);
  }
```

Figur 42 Kod före uppveckling av slinga.

```
  for(i=0; i<kk; i+=4) {
    standdev += (mo[i].muinf12- avemuinf)*(mo[i].muinf12-avemuinf)/(kk-1);
    standdev += (mo[i+1].muinf12- avemuinf)*(mo[i+1].muinf12-avemuinf)/(kk-1);
    standdev += (mo[i+2].muinf12- avemuinf)*(mo[i+2].muinf12-avemuinf)/(kk-1);
    standdev += (mo[i+3].muinf12- avemuinf)*(mo[i+3].muinf12-avemuinf)/(kk-1);
  }
```

Figur 43 Exempel på uppveckling av slinga.

4.16.2 Minnesoptimering av datastrukturen Mutobj

Ännu en minnesoptimering hittades i version 23 som berör minnesreservation med datastrukturen MutObj. Strukturen allokerade minne enligt maximalantalet aminosyrekombinationer i en sekvens eller N. Detta räknas ut med formeln i Figur 19.

Ömsesidig information räknas för maximalt N par aminosyror, detta antagande har redan konstaterats i tidigare versioner av programmet. Då kan det faktiska behovet av minnesreservering minskas med att reservera det antal som vi vet har passerat de kriterier som programmet körs med, eller variabeln kk. Om programmet körs med kommandot:

covana Herpes_MCP 0.21 0.04 0.2

kommer variabeln N att få värdet: 1 154 440 varav variabeln kk får värdet: 28 441.

Strukturen MutObj definieras i Figur 44, se nedan:

```
struct Mutobj {
  double muinf12;
  unsigned int site1;
  unsigned int site2;
};
```

Figur 44 Definition av strukturen Mutobj

Genom att se på strukturen går det att se att det finns två stycken unsigned integervärden samt en variabel av datatypen double vilket ger storleken 16 byte totalt. Optimeringen resulterade i att minnesallokeringen gick från:

16*1154440 = 18471040 byte eller 18 Mb, till

16*28441 = 455056 byte eller 0,455 Mb

4.17 Kontroll av data samt förbättring av implementationen

När en optimering blir utförd körs programmet och en kontroll ifall programmets utdata överrensstämmer med den utdata som det ursprungliga programmet skapade. Ifall någonting skiljer sig i utdata från version till version så betyder det att ett fel (bug på engelska) har introducerats samt att det behöver korrigeras. En sådan versions körtid samt minnesanvändning kan inte tas i beaktande som en optimering, däremot är dessa versioner viktiga för att komma vidare med att korrigera introducerade fel.

Här kan noteras att tidstagningen hoppar i versioneringen från version 15 till 24. De mellanliggande versionerna är deloptimeringar av en större helhet. En del versioner producerade felaktigt resultat och blev därför korrigerade i påföljande versioner. Versioner som producerar felaktiga data jämfört med det ursprungliga programmet kan inte tas med som ett resultat av en optimering. Detta är orsaken till varför versionsbeskrivningen inte är kontinuerlig.

Alla varningar som genererades i kompileringen blev korrigerade i version 25. Samt alla större matrisers minnesallokering ändrades från lös (Sparse Memory Allocation på engelska) till kontinuerlig (Continuous Memory Allocation på engelska). Detta betyder att minnet allokeras i en kontinuerlig klump istället för många små klumpar vilket borde inverka på minnesoperationerna. Att ha allt minne kontinuerligt kan

snabba upp minnesoperationer eftersom mera cacheträffar borde förekomma. Det totala programmets minnesanvändning minskades i optimeringen med 12 Mb från 73 Mb till 61 Mb, och körtiden minskade med 0,13 sekunder från 2,72 s till 2,59 s.

4.18.1 Kontinuerlig minnesallokering

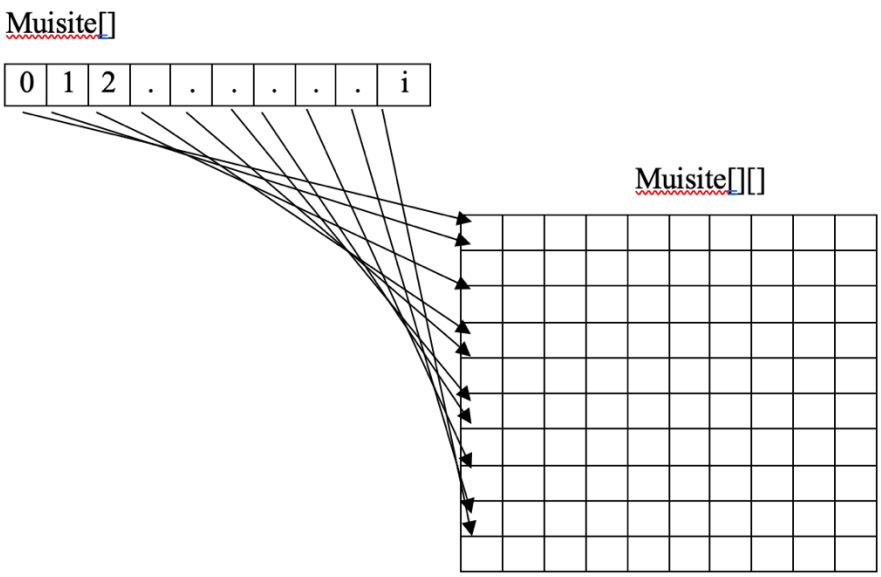
MuiSite datastrukturens räcka sitej[] minnesallokering sker nu enligt kontinuerlig minnesallokering (Continous Memory Allocation på engelska). Tidigare allokerades detta minne löst, det går också att kalla detta för (Sparse Memory Allocation på engelska). Dvs, minnesallokeringen utfördes i flera små etapper vilket kan leda till minnesfragmentering.

Ett exempel på lös minnesallokering visas nedan:

```
int** muisite;
try {
  muisite = new int*[kk];
  for (k=0; k<kk; k++)      muisite[k]= new int[kk];

} catch (...) {
  cout<<"Exception raised: muisite"<<"\n";
  return 1;
}
```

Figur 45 Ursprungliga programmets minnesallokering av muisite.



Figur 46 Illustration av gles minnesreservation

I figur 45 beskrivs hur matrisen muisite initialiseras med hjälp av en slinga. Först

initialiseras den första räckan till en integer räcka med variabeln kk's längd. Kk variabeln representerar i detta program det antal par aminosyror som har ett gemensamt ömsesidigt informationsvärde över den gräns som definierats i samband med körningen. Efter detta skede tilldelas nya räckor till varje element i räckan muisite[]. Detta görs i en slinga som går igenom alla element i den första räckan. För varje element allokeras en ny integerräcka med variabeln kk's längd. Allokeringen ovan sköts på ett sätt som kallas för lös minnesreservation (Sparse Memory Allocation på engelska).

Det är inte optimalt att i en slinga reservera minne enskillt för varje kolumn i en matris. Problemet med detta kan vara att minnesallokeringen reserverar platser i minnet på flera olika ställen. Istället för att reservera minnet i en kontinuerlig del av minnet. Det finns risk för minnesfragmentering när minnet reserveras på detta sätt. Problemet med att minnet är fragmenterat betyder att en minnesåtkomst kan behöva mera tid för att slutföras. Orsaken bakom detta är att minnescachen nödvändigtvis inte räcker till att spara alla registeradresser i minnet som t.ex. dubbelräckan muisite upptar.

Användningen av kontinuerlig minnesallokering (Continous Memory Allocation på engelska) eliminerar problemet med fragmentering av minnesanvändning. Programmet försöker reservera en kontinuerlig plats i minnet med hjälp av kommandot malloc(). Ett exempel av kontinuerlig minnesallokering kan ses nedan:

```
char **seq;
char *SEQ;

  seq = (char**) malloc(seqnum*sizeof(char*));
  SEQ = (char*)malloc(seqnum*resnum*sizeof(char));
```

Figur 47 Kontinuerlig minnesallokering.

4.18.2 Optimering av datastrukturen MuiSite

I version 25 förbättrades indexeringen i MuiSite matrisen så att räckans index MuiSite ms[index] används istället för att använda en skilld integerdatatyp. På detta sätt kan en integerdatatyp tas bort per element i räckan MuiSite. Detta sparar 4 byte per element i den första räckan med matrisen muisite. Idén uppstod efter att värden i matrisen MuiSite sorterats samt initialiserat längden på MuiSite räckan till samma

antal som antalet tecken i FASTA-filen, eller längden av aminosyresekvensen som representeras med variabeln resnum. Efter att koden studerats kunde första radens värden identifieras att egentligen vara positionerna för aminosyrorna i FASTA-filen. Positionsvärdena är ett index som endast varierar i längd beroende på aminosyresekvensens längd. Längden är ändå maximalt lika lång som räckan MuiSite[].

Alla ändringar som beskrivs i denna version minskade minnesanvändningen med 24 Mb där den totalt uppgår till 37 Mb. Körtiden i versionen är 2,03 sekunder.

4.19 Exakt uträkning av minnesanvändning

I version 26 räknas det exakta minnesbehovet ut för matriserna seq och repseq. Det ursprungliga programmet använde sig av konstanten MaxResN, detta ändrades till resnum vilket är det exakta antalet tecken per sekvens. MaxResN innehåller ett konstant värde på 5000, detta är det maximiantal tecken en sekvens får innehålla. Att istället använda variabeln resnum betyder mindre minnesanvändning. Från kapitel 5.1 mätresultat kan ses att version 26 befinner sig mellan versionerna 24 och 31 detta betyder att ändringen bidrog till en minnesminskning på 24 Mb. Märk väl att detta inte är den enda ändringen som resulterade i minnesminskningen på 24 Mb.

4.20 Ersättning av frekvent uträknade värden

I version 27 till 29 korrigerades fel i sökningen av tripletter. I version 29 introducerades en optimering där ett försök att eliminera de uträkningar som kan utföras en gång och sparas i minne, för att följande gånger endast hämta uträkningen ur minnet istället för att räkna om samma sak igen.

5. Resultat av optimeringar

Tre huvudsakliga mätningar gjordes i detta optimeringsarbete, hur lång tid, hur mycket minne samt hur mycket lagringsutrymme som behövs för en körning. Optimeringar utfördes och benchmarkades oftast en efter en men i vissa versioner har flera optimeringar blivit utförda och endast en tidtagning. I sådana fall där flera optimeringar blivit tidtagna tillsammans är det svårt att urskilja vilken optimering som orsakade den bästa förbättringen. I de versioner som flera optimeringar har gjorts, i samma version, har jag försökt implementera en och samma typ av optimering. I de fall där detta misslyckats har implementationen hindrat mig från att på ett enkelt sätt kunna införa optimeringen utan att beröra flera delar av koden. I dessa situationer kan också versionsnummer hoppa från t.ex. version 15 till version 24. Luckor i versionerna innebär oftast att de mellanliggande versionerna inte producerade korrekt resultat eller endast fungerat som deltester av en större optimeringshelhet.

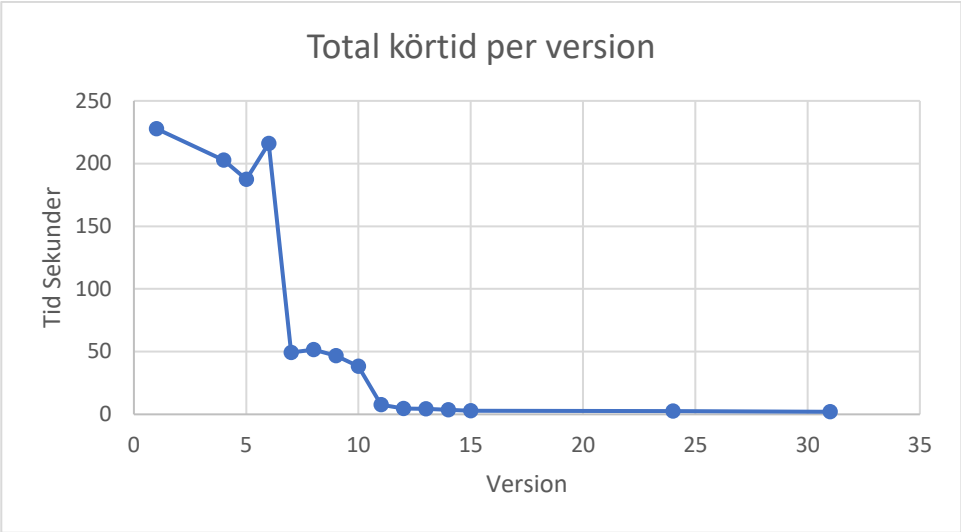
I kapitel 5.1 visas mätresultaten som körtid, minnesanvändning samt lagringsutrymme i en tabell och visualiserade i grafer.

5.1 Mätresultat

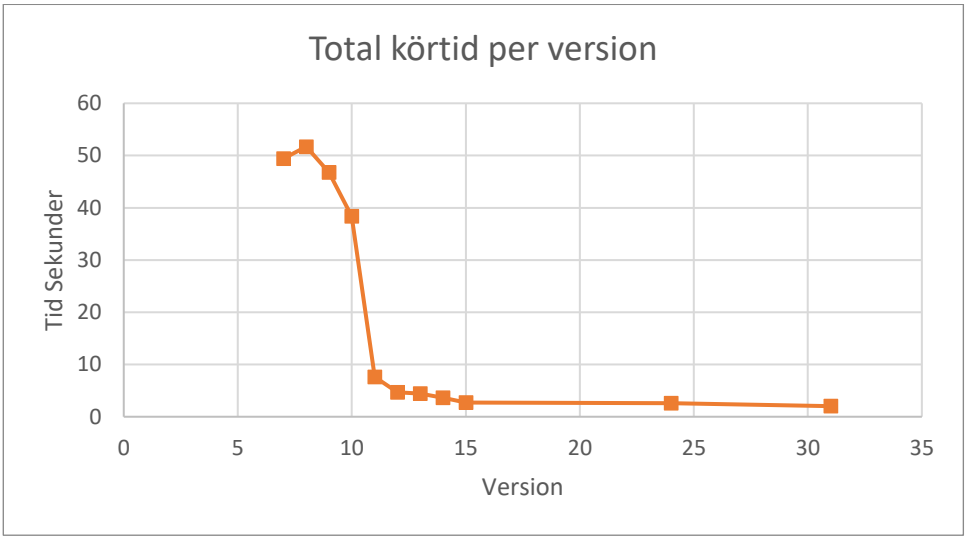
Programmets exekveringskommando var följande:
./covana Herpes_MCP 0.21 0.04 0.2

Version	Total körtid (s)	Minnes användning (Mb)	Diskutrymme (Mb)
1	227,83	3250	277
4	202,67	110	277
5	187,5	110	277
6	216	110	277
7	49,4	68	277
8	51,7	68	277
9	46,8	68	277
10	38,4	68	168
11	7,65	68	21
12	4,7	72	21
13	4,45	72	21
14	3,64	72	21
15	2,72	73	21
24	2,59	61	21
31	2,03	37	21

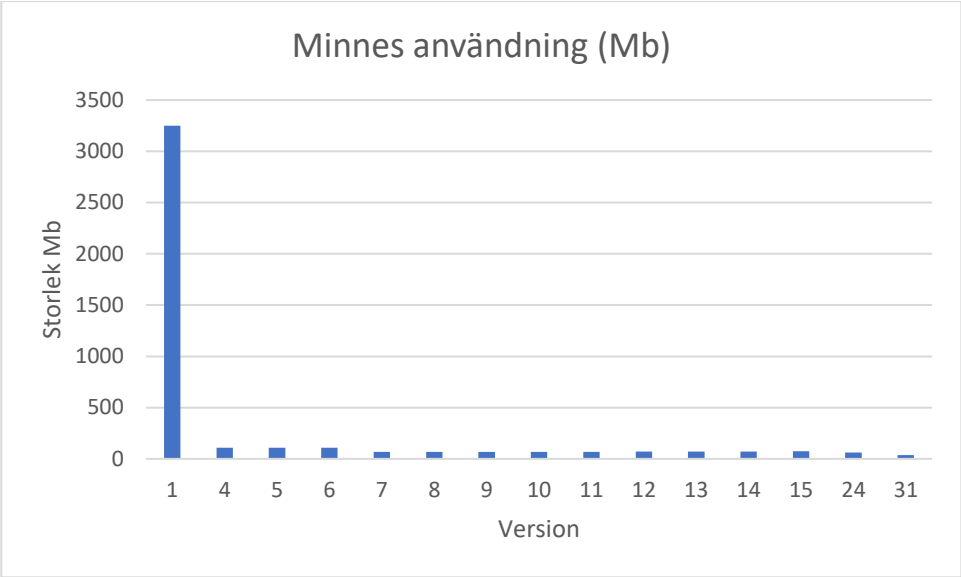
Tabell 2 Mätdata per optimerad version.



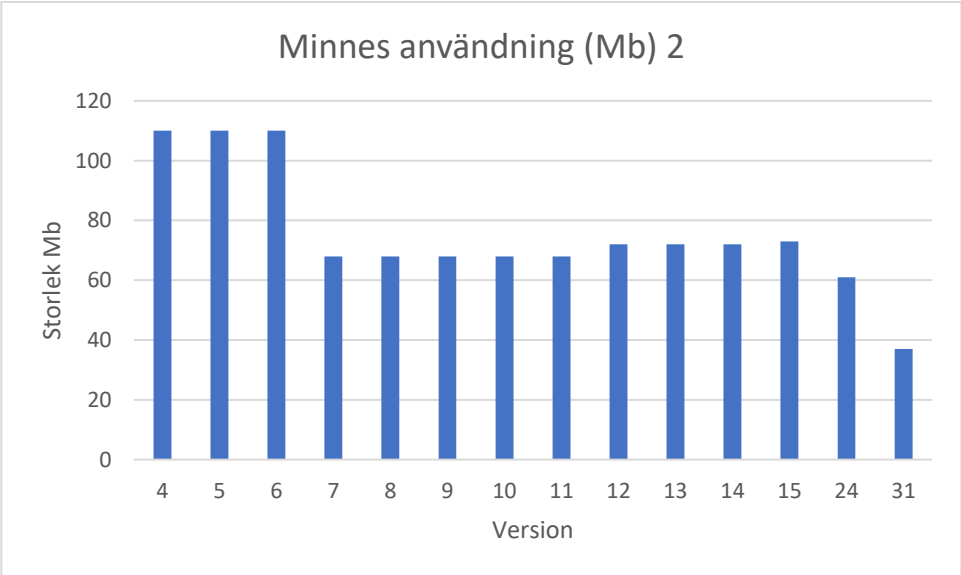
Figur 48 Total körtid per version.



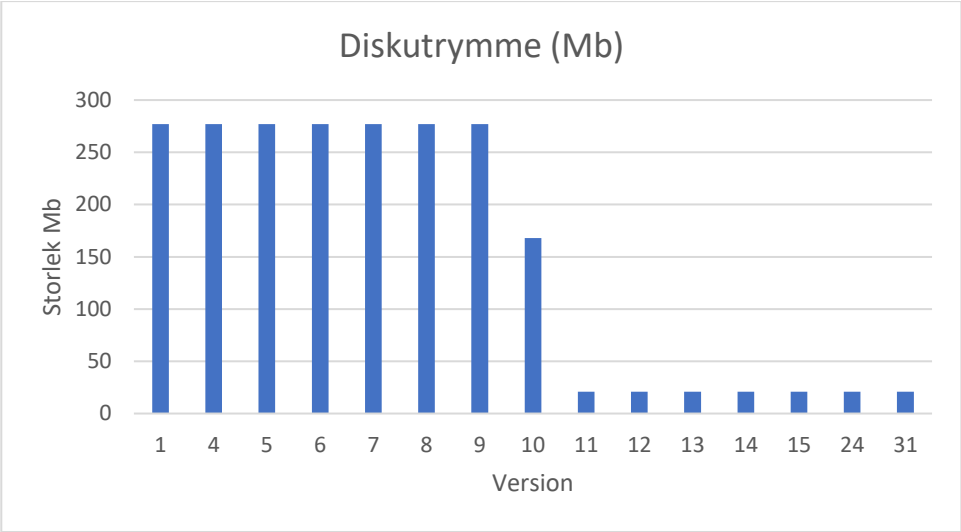
Figur 49 Total körtid per version – uppförstorad genom att skippa de första 5 versionerna.



Figur 50 Minnesanvändning per version.



Figur 51 – Minnesanvändning uppförstorad genom att hoppa över version 1.



Figur 52 Graf över skivutrymmesanvändning per version.

6. Avslutning

De optimeringar som beskrivits i detta diplomarbete behandlar många områden. Det finns flera saker som behöver tas i beaktande för att optimera ett program så att det kör snabbt och använder så litet RAM-minne och diskutrymme som möjligt. Redan enkla definitioner av datastrukturer som kommer att användas i olika algoritmer påverkar hur mycket minne som används. Till exempel en unsigned integer eller short, beskriver endast positiva heltal men använder hälften av minnet jämfört med en signed integer.

Hur data behandlas med hjälp av datastrukturer är också mycket viktigt för prestandan. Till exempel om den data som sparas i räckor är sorterad innebär det att effektiva sökalgoritmer kan användas istället för mindre effektiva på en osorterad mängd data. Även på det sätt som data skrivs till filsystemet påverkar programmets prestanda genom att separera filskrivningen från tidskritiska delar där det kan påverka processor och minnesoperationer.

Som resultat av alla optimeringar påverkades programmets prestanda positivt genom algoritmoptimeringar inom triplettanalys, ömsesidig informationsanalys samt filhanteringsalgoritmer.

Den viktigaste optimeringen som gjordes påverkade analys av triplettinformation. Här minskades minnesanvändningen med en faktor på ~87 gånger. Detta var den första flaskhals som märktes och som förbättrades så att andra optimeringar kunde utföras därefter. Den följande effektivitetsoptimeringen i storleksordning innebar en ändring hur den information som skrevs till filer hanterades. Istället för att skriva många filer samlades informationen ihop till en och samma fil. Genom att också minska på antalet tecken för att representera ett mätvärde minskades storleken på den data som behöver sparas.

Hur programmet beter sig vid olika storlekar av fasta-filer kunde vidare studeras i samband med detta arbete. Detta har inte behandlats på djupet, den fil som nu använts består av 1520 tecken per sekvens och sekvensantaler är 90.

En annan sak som kunde vara värt att studera är att använda datorns GPU istället för, eller i kombination med, datorns CPU. En artikel om FastGCN som hittas i referens [24] beskrivs en studie om hur effektivt en GPU kan användas istället för en CPU. Här har resultatet visat att analyser med stor mängd data är effektivare med en GPU implementation. Detta kunde eventuellt också förbättra prestandan på detta program med det har inte studerats i detalj.

Det antagande som gjordes i början av arbetet att eventuellt använda sig av MPI [15] kunde avskrivas efter att minneskravet minskades så kraftigt att det inte längre var ett problem att köra programmet på endast en dator i stället för många datorer. Slutresultatet, av alla optimeringar som är beskrivna i detta arbete, blev att det optimerade programmet kunde köras 112 gånger snabbare med en minnesanvändning från 3,2 Gb till 37 Mb och diskutrymme från 277 Mb till 21 Mb.

Litteraturförteckning

[1] André Norrgård, Kandidatavhandling, Databehandling inom protein-, dna- och aminosyreanalyser.

[2] ProCon, Webapplikation för visualisering av proteinkonservation, (Senast läst: 18.12.2018) http://structure.bmc.lu.se/ProCon/

[3] Robert Sedgewick, Algorithms in C, Third Edition, Addison-Wesley 1998, ISBN: 0201314525

[4] ProCon algoritmbeskrivning, websida med URL: http://structure.bmc.lu.se/ProCon/algorithm.shtml

[5] Stanley B. Lippman, Josée Lajoie, Barbara E. Moo. C++ Primer, Fourth Edition. ISBN 0-201-72148-1

[6] Gentekniknämnden, 2018, Online referens, https://genteknik.nu/aminosyra/.

[7] Protein, https://www.collinsdictionary.com/dictionary/english/protein

[8] Bairong Shen, Mauno Vihinen; Conservation and covariance in PH domain sequences: physicochemical profile and information theoretical analysis of XLA-causing mutations in the Btk PH domain, *Protein Engineering, Design and Selection*, Volume 17, Issue 3, 1 March 2004, Pages 267–276, https://doi.org/10.1093/protein/gzh030

[9] cplusplus.com Referens till c++ bibliotek stdlib.h., Quick Sort algoritm. URL: http://www.cplusplus.com/reference/cstdlib/qsort/.

[10] Ursprungliga programmet Covana.cpp, skrivet av Bairong Shen.

[11] Software Optimization Guide for AMD Athlon™ 64 and AMD Opteron™ Processors, 2001 – 2003 Advanced Micro Devices, Inc. Rev. 3.03 September 2003

[12] Beals, M., Gross, L. & Harrell, S. *Amino Acid Frequency*. (1999) http://www.tiem.utk.edu/~gross/bioed/webmodules/aminoacid.htm. (Senast läst: 4.12.2018)

[13] Bairong Shen, Självbiografi, https://systemsbiology.org/bio/bairong-shen/ (Senast läst: 11.12.2018)

[14] Mauno Vihinen, Självbiografi, URL: http://portal.research.lu.se/portal/en/persons/mauno-vihinen(effb5f9d-8a46-4a53-8112-801680afcae3).html#Overview (Senast läst: 11.12.2018)

[15] Message Passing Interface, https://www.mpi-forum.org/ (Last read: 11.12.2018)

[16] Profileringsverktyget gprof, https://sourceware.org/binutils/docs/gprof/ (Senast läst: 11.12.2018)

[17] Business Finland, (Senast läst: 11.12.2018) https://www.businessfinland.fi/sv/for-finlandska-kunder/framsida/

[18] Seagate, Desktop HDD, SATA Product Manual, PDF https://www.seagate.com/www-content/product-content/desktop-hdd-fam/en-us/docs/100782401c.pdf

[19] Kingston HyperX, Memory Module Specifications, HX432C16PB3AK2/16 https://www.kingston.com/dataSheets/HX432C16PB3AK2_16.pdf

[20] Sumedh N. (Intel),Coding for Performance:Data alignment and structures, https://software.intel.com/en-us/articles/coding-for-performance-data-alignment-and-structures

[21] Michael Sipser, Introduction to the theory of Computation, PWS Publishing Company, ISBN: 0-534-94728-X

[22] Nätkurs i Computational Geometry, Online Web Page URL: https://sites.google.com/site/d7013e11/, senast last: 5.5.2019

[23] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwartzkopf, Computational Geometry: Algorithms and Applications, Third Edition, Springer, ISBN: 978-3-540-77973-5

[24] Liang M, Zhang F, Jin G, Zhu J (2015) FastGCN: A GPU Accelerated Tool for Fast Gene Co-Expression Networks. PLoS ONE 10(1): e0116776. https://doi.org/10.1371/journal.pone.0116776, senast läst: 04.09.2019.